笨办法学 Python

Release 2.0

作者：Zed A. Shaw, 翻译：Wang Dingwei

March 12, 2012

VISIT…

LANZAROTE

Caliente.com

CONTENTS

译者前言 **3**

前言：笨办法更简单 **5**
- 读和写 . . . 5
- 注重细节 . . . 5
- 发现不同 . . . 6
- 不要复制粘贴 . . . 6
- 对于坚持练习的一点提示 . . . 6
- 给“小聪明”们的警告 . . . 7
- 许可协议 . . . 7
- 特别感谢 . . . 7

习题 0: 准备工作 **9**
- Mac OSX . . . 9
- Windows . . . 10
- Linux . . . 12
- 给新手的告诫 . . . 14

习题 1: 第一个程序 **15**
- 你应该看到的内容 . . . 15
- 加分习题 . . . 16

习题 2: 注释和井号 **19**
- 你应该看到的结果 . . . 19
- 加分习题 . . . 19

习题 3: 数字和数学计算 **21**
- 你应该看到的结果 . . . 22
- 加分习题 . . . 22

习题 4: 变量 (variable) 和命名 **23**

你应该看到的结果 24
加分习题 24

习题 5: 更多的变量和打印 **25**
你应该看到的结果 26
加分习题 26

习题 6: 字符串 (string) 和文本 **27**
你应该看到的结果 28
加分习题 28

习题 7: 更多打印 **29**
你应该看到的结果 29
加分习题 30

习题 8: 打印，打印 **31**
你应该看到的结果 31
加分习题 31

习题 9: 打印，打印，打印 **33**
你应该看到的结果 33
加分习题 34

习题 10: 那是什么？ **35**
你应该看到的结果 36
加分习题 36

习题 11: 提问 **37**
你应该看到的结果 37
加分习题 38

习题 12: 提示别人 **39**
你应该看到的结果 39
加分习题 39

习题 13: 参数、解包、变量 **41**
等一下！“功能”还有另外一个名字 41
你应该看到的结果 42
加分习题 43

习题 14: 提示和传递 **45**
你应该看到的结果 45
加分习题 46

习题 15: 读取文件 **47**
你应该看到的结果 48

加分习题 . . . 48

习题 16: 读写文件 51
你应该看到的结果 . . . 52
加分习题 . . . 52

习题 17: 更多文件操作 55
你应该看到的结果 . . . 56
加分习题 . . . 56

习题 18: 命名、变量、代码、函数 57
你应该看到的结果 . . . 58
加分习题 . . . 59

习题 19: 函数和变量 61
你应该看到的结果 . . . 62
加分习题 . . . 62

习题 20: 函数和文件 63
你应该看到的结果 . . . 64
加分习题 . . . 64

习题 21: 函数可以返回东西 65
你应该看到的结果 . . . 66
加分习题 . . . 66

习题 22: 到现在你学到了哪些东西? 69
你学到的东西 . . . 69

习题 23: 读代码 71

习题 24: 更多练习 73
你应该看到的结果 . . . 74
加分习题 . . . 74

习题 25: 更多更多的练习 75
你应该看到的结果 . . . 76
加分习题 . . . 77

习题 26: 恭喜你，现在可以考试了! 79

习题 27: 记住逻辑关系 81
逻辑术语 . . . 81
真值表 . . . 82

习题 28: 布尔表达式练习 85

你应该看到的结果 . . . 86
加分习题 . . . 87

习题 29: 如果 (if) **89**
你应该看到的结果 . . . 90
加分习题 . . . 90

习题 30: Else 和 If **91**
你应该看到的结果 . . . 92
加分习题 . . . 92

习题 31: 作出决定 **93**
你应该看到的结果 . . . 94
加分习题 . . . 95

习题 32: 循环和列表 **97**
你应该看到的结果 . . . 98
加分习题 . . . 99

习题 33: While 循环 **101**
你应该看到的结果 . . . 102
加分习题 . . . 102

习题 34: 访问列表的元素 **105**
加分习题 . . . 106

习题 35: 分支和函数 **107**
你应该看到的结果 . . . 109
加分习题 . . . 109

习题 36: 设计和调试 **111**
If 语句的规则 . . . 111
循环的规则 . . . 111
调试 (debug) 的小技巧 . . . 112
家庭作业 . . . 112

习题 37: 复习各种符号 **113**
Keywords（关键字） . . . 113
数据类型 . . . 114
字符串转义序列 (Escape Sequences) . . . 115
字符串格式化 (String Formats) . . . 115
操作符号 . . . 116

习题 38: 阅读代码 **119**
加分习题 . . . 119

习题 39: 列表的操作 121
你应该看到的结果 122
加分习题 123

习题 40: 字典, 可爱的字典 125
你应该看到的结果 127
加分习题 127

习题 41: 来自 Percal 25 号行星的哥顿人 (Gothons) 129
你应该看到的结果 134
加分习题 136

习题 42: 物以类聚 137
你应该看到的结果 142
加分习题 143

习题 43: 你来制作一个游戏 145

习题 44: 给你的游戏打分 147
函数的风格 147
类的风格 147
代码风格 148
好的注释 148
为你的游戏评分 149

习题 45: 对象、类、以及从属关系 151
写成代码是什么样子 152

习题 46: 一个项目骨架 155
骨架内容 155
测试你的配置 157
使用这个骨架 157
小测验 157

练习 47: 自动化测试 159
撰写测试用例 159
测试指南 161
你应该看到的结果 161
加分习题 161

习题 48: 更复杂的用户输入 163
我们的游戏语汇 163
你应该测试的东西 165
设计的技巧 166
加分习题 167

习题 49: 创建句子 **169**
匹配 (Match) 和窥视 (Peek) . . . 169
句子的语法 . . . 170
关于异常 (Exception) . . . 172
你应该测试的东西 . . . 173
加分习题 . . . 173

习题 50: 你的第一个网站 **175**
安装 lpthw.web . . . 175
写一个简单的"Hello World"项目 . . . 176
发生了什么事情? . . . 177
修正错误 . . . 177
创建基本的模板文件 . . . 178

习题 51: 从浏览器中获取输入 **181**
Web 的工作原理 . . . 181
表单 (form) 的工作原理 . . . 183
创建 HTML 表单 . . . 184
创建布局模板 (layout template) . . . 186
为表单撰写自动测试代码 . . . 187
加分习题 . . . 189

习题 52: 创建你的 web 游戏 **191**
重构《习题 42》的游戏 . . . 191
会话 (session) 和用户跟踪 . . . 196
创建引擎 . . . 197
你的期末考试 . . . 200

下一步 **201**

老程序员的建议 **203**

Indices and tables **205**

Contents:

译者前言

《笨办法学 Python》(Learn Python The Hard Way，简称 LPTHW) 是 Zed Shaw 编写的一本 Python 入门书籍。适合对计算机了解不多，没有学过编程，但对编程感兴趣的朋友学习使用。这本书以习题的方式引导读者一步一步学习编程，从简单的打印一直讲到完整项目的实现。也许读完这本书并不意味着你已经学会了编程，但至少你会对编程语言以及编程这个行业有一个初步的了解。

本书区别于其它入门书籍的特点如下：

- 注重实践。本书提供了足够的练习代码，如果你完成了所有的练习（包括加分习题），那你已经写了上万行的代码。要知道很多职业程序员一年也就写几万行代码而已。
- 注重能力培养。除了原序言提到的“读和写”、“注重细节”、以及“发现不同”这样的基本能力以外，本书还培养了读者自己专研问题和寻求答案的能力。
- 注重好习惯的养成。本书详细地讲解了怎样写出好的代码、好的注释、好的项目。这会让你在后续的学习中少走很多弯路。

本书结构非常简单，其实就是 52 个习题。其中 26 个覆盖了输入输出、变量、以及函数三个课题，另外 26 个覆盖了一些比较高级的话题，如条件判断、循环、类和对象、代码测试、以及项目的实现等。每一章节的格式基本都是一样的，以代码练习题开始，读者照着说明编写代码（不允许复制粘贴），运行并检查结果，然后再做一下加分习题就可以了。当然如果你觉得加分习题对你来说有点难，你也可以暂时跳过，以后再完成也没关系。

另外阅读本书还需要你有一定的英文能力。其实学编程不懂英语是很吃亏的，毕竟编程语言都是基于英语，而编程社群的主要交流方式也是英语。不会英语的人在编程界可能就只好当二等公民了。本书的翻译尽量保留了所有的英文专业词汇（可能会有中文说明），而且遵照 Zed 的建议，代码及答案部分没有翻译成中文，读者看到不懂的地方，请自己查字典解决。

如果你对自己的英文能力比较有信心，译者强烈推荐你直接去下载阅读英文原版。这本书代码较多，文字内容较少，因此英文原版的阅读理解也比较容易。

LPTHW 的风格和别的书差异很大。它没有像一般的入门书籍一样通过讨好读者以激发读者兴趣，而是直截了当地告诉你你需要做什么，需要注意什么。这种风格可能会让人觉得枯燥乏味，读者姑且把这也当做 Hard Way 的一部分把。所以如果你觉得实在不能适应这种风格，Zed 推荐你看下面两本书:

- How To Think Like A Computer Scientist
- A Byte Of Python 这本书有 中译版

本书的电子版会随时跟着作者更新。你可以通过 Read The Docs 读到最新的网页版内容，也可以到 bitbucket 代码仓库下载 PDF 文件。如果你对本书的翻译有任何意见和建议，你可以通过 bitbucket 进行反馈。

你可以访问 lulu.com 购买本书的英文印刷版，这也是对原作者的支持。

原书版权为 Zed Shaw 所有，译文版权为 Zed Shaw 和译者共有。译文遵循原书的版权规定：只允许完整转载，禁止商业用途。

前言：笨办法更简单

这本小书的目的是让你起步编程。虽然书名说是“笨办法”, 但其实并非如此. 所谓的“笨办法”是指本书教授的方式。在这本书的帮助下，你将通过非常简单的练习学会一门编程语言。做练习是每个程序员的必经之路：

1. 做每一道习题
2. 一字不差地写出每一个程序
3. 让程序运行起来

就是这样了。刚开始这对你来说会非常难，但你需要坚持下去。如果你通读了这本书，每晚花个一两小时做做习题，你可以为自己读下一本编程书籍打下良好的基础。通过这本书你学到的可能不是真正的编程，但你会学到最基本的学习方法。

这本书的目的是教会你编程新手所需的三种最重要的技能：读和写、注重细节、发现不同。

读和写

很显然，如果你连打字都成问题的话，那你学习编程也会成问题。尤其如果你连程序源代码中的那些奇怪字符都打不出来的话，就根本别提编程了。没有这样基本技能的话，你将连最基本的软件工作原理都难以学会。

为了让你记住各种符号的名字并对它们熟悉起来，你需要将代码写下来并且运行起来。这个过程也会让你对编程语言更加熟悉。

注重细节

区分好程序员和差程序员的最重要的一个技能就是对于细节的注重程度。事实上这是任何行业区分好坏的标准。如果缺乏对于工作的每一个微小细节的注意，你的工作成果将缺乏重要的元素。以编程来讲，这样你得到的结果只能是毛病多多难以使用的软件。

通过将本书里的每一个例子一字不差地打出来，你将通过实践训练自己，让自己集中精力到你作品的细节上面。

发现不同

程序员长年累月的工作会培养出一个重要技能，那就是对于不同点的区分能力。有经验的程序员拿着两份仅有细微不同的程序，可以立即指出里边的不同点来。程序员甚至造出工具来让这件事更加容易，不过我们不会用到这些工具。你要先用笨办法训练自己，等你具备一些相关能力的时候才可以使用这些工具。

在你做这些练习并且打字进去的时候，你一定会写错东西。这是不可避免的，即使有经验的程序员也会偶尔写错。你的任务是把自己写的东西和要求的正确答案对比，把所有的不同点都修正过来。这样的过程可以让你对于程序里的错误和 bug 更加敏感。

不要复制粘贴

你必须手动将每个练习打出来。复制粘贴会让这些练习变得毫无意义。这些习题的目的是训练你的双手和大脑思维，让你有能力读代码、写代码、观察代码。如果你复制粘贴的话，那你就是在欺骗自己，而且这些练习的效果也将大打折扣。

对于坚持练习的一点提示

在你通过这本书学习编程时，我正在学习弹吉他。我每天至少训练 2 小时，至少花一个小时练习音阶、和声、和琶音，剩下的时间用来学习音乐理论和歌曲演奏以及训练听力等。有时我一天会花 8 个小时来练习，因为我觉得这是一件有趣的事情。对我来说，要学好一样东西，每天的练习是必不可少的。就算这天个人状态很差，或者说学习的课题实在太难，你也不必介意，只要坚持尝试，总有一天困难会变得容易，枯燥也会变得有趣了。

在你通过这本书学习编程的过程中要记住一点，就是所谓的“万事开头难”，对于有价值的事情尤其如此。也许你是一个害怕失败的人，一碰到困难就想放弃。也许你是一个缺乏自律的人，一碰到“无聊”的事情就不想上手。也许因为有人夸你“有天分”而让你自视甚高，不愿意做这些看上去很笨拙的事情，怕有负你”神童”的称号。也许你太过激进，把自己跟有 20 多年经验的编程老手相比，让自己失去了信心。

不管是什么原因，你一定要坚持下去。如果你碰到做不出来的加分习题，或者碰到一节看不懂的习题，你可以暂时跳过去，过一阵子回来再看。只要坚持下去，你总会弄懂的。

一开始你可能什么都看不懂。这会让你感觉很不舒服，就像学习人类的自然语言一样。你会发现很难记住一些单词和特殊符号的用法，而且会经常感到很迷茫，直到有一天，忽然一下子你会觉得豁然开朗，以前不明白的东西忽然就明白了。如果你坚持练习下去，坚持去上下求索，你最终会学会这些东西的。也许你不会成为一个编程大师，但你至少会明白程序是怎么工作的。

如果你放弃的话，你会失去达到这个程度的机会。你会在第一次碰到不明白的东西时 (几乎是所有的东西) 放弃。如果你坚持尝试，坚持写习题，坚持尝试弄懂习题的话，你最终一定会明白里边的内容的。

如果你通读了这本书，却还是不知道编程是怎么回事。那也没关系，至少你尝试过了。你可以说你已经尽过力但成效不佳，但至少你尝试过了。这也是一件值得你骄傲的事情。

给“小聪明”们的警告

有的学过编程的人读到这本书，可能会有一种被侮辱的感觉。其实本书中没有任何要居高临下地贬低任何人的意思。只不过是我比我面向的读者群知道的更多而已。如果你觉得自己比我聪明，然后觉得我在居高临下，那我也没办法，因为你根本就不属于我的目的读者群。

如果你觉得这本书里到处都在侮辱你的智商，那我对你有三个建议:

1. 别读这本书了。我不是写给你的，我是写给需要学习的人的。
2. 放下架子好好学。如果你认为你什么都知道，那你就很难从比你强的人身上学到什么了。
3. 学 Lisp 去。我听说什么都知道的人可喜爱 Lisp 了。

对于其他在这里学习的人，你们读的时候就想着我在微笑就可以了，虽然我的眼睛里还带着恶作剧的闪光。

许可协议

Copyright (C) 2010 by Zed A. Shaw. 你可以在不收取任何费用，而且不修改任何内容的前提下自由分发这本书给任何人。但是本书的内容只允许完整原封不动地进行分发和传播。也就是说如果你用这本书给人上课，只要你不向学生收费，而且给他们看的书是完整未加修改的，那就没问题。

特别感谢

首先我要感谢帮助我完成这版书的人。首先是 Pretty Girl Editing Services 可爱的编辑所做的编辑工作。然后是 Greg Newman，他提供了美工图并帮我设计了封面，而且还帮忙复审了本书。是他让这本书看上去像本真正的书籍，而且就算我没在第一版里提到他的辛劳，他也没跟我计较。我还要感谢 Brian Shumate 在网站设计方面的帮助，这方面的帮助也是我非常需要的。

最后，我还要感谢成千上万读过本书第一版而且提出 bug 报告和改进建议的读者。你们的贡献让这本书的内容更为扎实，没有你们我是做不到的。谢谢你们。

习题 0：准备工作

这道习题并没有代码内容，它的主要目的是让你在计算机上安装好 Python。你应该尽量照着说明进行操作，例如 Mac OSX 默认已经安装了 Python 2，所以就不要在上面安装 Python 3 或者别的 Python 版本了。

> **Warning:** 如果你不知道怎样使用 Windows 下的 PowerShell，或者 OSX 下的 Terminal，或者 Linux 下的"bash"，那你就需要学习了。我有一个免费的快速入门教程放在 http://cli.learncodethehardway.org/，你可以快速学到 PowerShell 和 Terminal 的基本用法。学完后再回来看这本书吧。

Mac OSX

你需要做下列任务来完成这个练习：

1. 用浏览器打开 http://learnpythonthehardway.org/exercise0.html 下载并安装 `gedit` 文本编辑器。
2. **把 gedit (也就是你的编辑器) 放到 Dock 中，以方便日后使用。**
    (a) 运行 gedit，我们要先改掉一些愚蠢的默认设定。
    (b) 从 `gedit menu` 中打开 `Preferences`，选择 `Editor` 页面。
    (c) 将 `Tab width:` 改为 4。
    (d) 选择 (确认有勾选到该选项) `Insert spaces instead of tabs`。
    (e) 然后打开"Automatic indentation"选项。
    (f) 转到 `View` 页面，打开"Display line numbers"选项。
3. 找到系统中的"命令行终端 (Terminal)"程序。到处找找，你会找到的。
4. 把 Terminal 也放到 Dock 里面。
5. 运行 Terminal 程序，这个程序看上去不怎么地。
6. 在 Terminal 程序里边运行 `python`。运行的方法是输入程序的名字再敲一下回车。
7. 敲击 CTRL-D (^D) 退出 python。

8. 这样你就应该退回到敲 `python` 前的提示界面了。如果没有的话自己研究一下为什么。
9. 学着使用 Terminal 创建一个目录，你可以上网搜索怎样做。
10. 学着使用 Terminal 进入一个目录，同样你可以上网搜索。
11. 使用你的编辑器在你进入的目录下建立一个文件。你将建立一个文件。使用“Save”或者“Save As…”选项，然后选择这个目录。
12. 使用键盘切换回到 Terminal 窗口，如果不知道怎样使用键盘切换，你一样可以上网搜索。
13. 回到 Terminal，看看你能不能使用命令看到你新建的文件，上网搜索如何将文件夹中的内容列出来。

Note: 安装 gedit 可能会遇到问题，对于非英文键盘布局的系统尤其如此。如果你碰到了问题，我建议你试试 Textwrangler，下载地址是 http://www.barebones.com/products/textwrangler/。

OSX: 你应该看到的结果

以下是我在自己电脑的 Terminal 中执行上述练习时看到的内容。和你做的结果会有一些不同，所以看看你能不能找出两者不同点来。

```
Last login: Sat Apr 24 00:56:54 on ttys001
~ $ python
Python 2.5.1 (r251:54863, Feb  6 2009, 19:02:12)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> ^D
~ $ mkdir mystuff
~ $ cd mystuff
mystuff $ ls
# ... 使用 gedit 编辑 text.txt ...
mystuff $ ls
test.txt
mystuff $
```

Windows

Note: Contributed by zhmark.

---

1. 用浏览器打开 http://learnpythonthehardway.org/exercise0.html 下载并安装 `gedit` 文本编辑器。这个操作无需管理员权限。

2. **把 `gedit` 放到桌面或者快速启动栏，这样你就可以方便地访问到该程序了。这两条在安装选项中可以看到。**
    (a) 运行 gedit，我们要先改掉一些愚蠢的默认设定。
    (b) 从 `gedit menu` 中打开 `Preferences`，选择 `Editor` 页面。
    (c) 将 `Tab width:` 改为 4。
    (d) 选择 (确认有勾选到该选项) `Insert spaces instead of tabs`。
    (e) 然后打开 "Automatic indentation" 选项。
    (f) 转到 `View` 页面，打开 "Display line numbers" 选项。
3. 从开始菜单运行 "PowerShell" 程序。你可以使用开始菜单的搜索功能，输入名称后敲回车即可打开。
4. 为它创建一个快捷方式，放到桌面或者快速启动栏中以方便使用。
5. 运行 Terminal 程序，这个程序看上去不怎么地。
6. **在 Terminal 程序里边运行 `python`。运行的方法是输入程序的名字再敲一下回车。**
    (a) 如果你运行 `python` 发现它不存在 (系统找不到 `python` 云云)。你需要访问 http://python.org/download 并且安装 Python。
    (b) 确认你安装的是 Python 2 而不是 Python 3。
    (c) 你也可以试试 ActiveState Python，尤其是你没有管理员权限的时候。
    (d) 如果你安装好了但是 `python` 还是不能被识别，那你需要在 powershell 下输入并执行以下命令：

    ```
    [Environment]::SetEnvironmentVariable("Path", "$env:Path;C:\Python27",
    "User")
    ```

    (e) 关闭并重启 powershell，确认 python 现在可以运行。如果不行的话你可能需要重启电脑。
7. 键入 CTRL-Z (^Z)，再敲回车以退出 `python`。
8. 这样你就应该退回到敲 `python` 前的提示界面了。如果没有的话自己研究一下为什么。
9. 学着使用 Terminal 创建一个目录，你可以上网搜索怎样做。
10. 学着使用 Terminal 进入一个目录。同样你可以上网搜索。
11. 使用你的编辑器在你进入的目录下建立一个文件。你将建立一个文件，使用 "Save" 或者 "Save As..." 选项，然后选择这个目录。
12. 使用键盘切换回到 Terminal 窗口，如果不知道怎样使用键盘切换，你一样可以上网搜索。
13. 回到 Terminal，看看你能不能使用命令看到你新建的文件，上网搜索如何将文件夹中的内容列出来。

> **Warning:** 有时这一步你会漏掉：Windows 下装了 Python 但是没有正确配置路径。确认你在 powershell 下输入了 `[Environment]::SetEnvironmentVariable("Path", "$env:Path;C:\Python27", "User")`。你也许需要重启 powershell 或者计算机来让路径设置生效。

Windows: 你应该看到的结果

```
> python
ActivePython 2.6.5.12 (ActiveState Software Inc.) based on
Python 2.6.5 (r265:79063, Mar 20 2010, 14:22:52) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> ^Z


> mkdir mystuff

> cd mystuff

... Here you would use gedit to make test.txt in mystuff ...

>
   <bunch of unimportant errors if you istalled it as non-admin - ignore them - hit Enter>
> dir
 Volume in drive C is
 Volume Serial Number is 085C-7E02

 Directory of C:\Documents and Settings\you\mystuff

04.05.2010  23:32    <DIR>          .
04.05.2010  23:32    <DIR>          ..
04.05.2010  23:32                 6 test.txt
               1 File(s)              6 bytes
               2 Dir(s)  14 804 623 360 bytes free

>
```

你看到的命令行信息，Python 信息，以及其它一些东西可能会非常不一样，不过应该大致不差。你可以通过 http://learnpythonthehardway.org 把你找到的错处告诉我们，我们会修正过来。

Linux

Linux 系统可谓五花八门，安装软件的方式也各有不同。我们假设作为 Linux 用户的你已经知道如何安装软件包了，以下是给你的操作说明：

1. (a) 用浏览器打开 http://learnpythonthehardway.org/exercise0.html 下载并安装 gedit 文本编辑器。
2. **把 gedit (也就是你的编辑器) 放到窗口管理器显见的位置，以方便日后使用。**
    (a) 运行 gedit，我们要先改掉一些愚蠢的默认设定。
    (b) 从 gedit menu 中打开 Preferences，选择 Editor 页面。

   (c) 将 `Tab width:` 改为 4。

   (d) 选择 (确认有勾选到该选项) `Insert spaces instead of tabs`。

   (e) 然后打开“Automatic indentation”选项。

   (f) 转到 `View` 页面，打开 “Display line numbers” 选项。

3. 找到 “Terminal” 程序。它的名字可能是 `GNOME Terminal`、`Konsole`、或者 `xterm`。
4. 把 Terminal 也放到 Dock 里面。
5. 运行 Terminal 程序，这个程序看上去不怎么地。
6. 在 Terminal 程序里边运行 `python`。运行的方法是输入程序的名字再敲一下回车。a. 如果你运行 `python` 发现它不存在的话，你需要安装它，而且要确认你安装的是 Python 2 而非 Python 3。
7. 敲击 CTRL-D (^D) 以退出 `python`。
8. 这样你就应该退回到敲 `python` 前的提示界面了。如果没有的话自己研究一下为什么。
9. 学着使用 Terminal 创建一个目录。你可以上网搜索怎样做。
10. 学着使用 Terminal 进入一个目录。同样你可以上网搜索。
11. 使用你的编辑器在你进入的目录下建立一个文件。你将建立一个文件，使用“Save”或者“Save As...”选项，然后选择这个目录。
12. 使用键盘切换回到 Terminal 窗口，如果不知道怎样使用键盘切换，你一样可以上网搜索。
13. 回到 Terminal，看看你能不能使用命令看到你新建的文件，上网搜索如何将文件夹中的内容列出来。

Linux: 你应该看到的结果

```
[~]$ python
Python 2.6.5 (r265:79063, Apr  1 2010, 05:28:39)
[GCC 4.4.3 20100316 (prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
[~]$ mkdir mystuff
[~]$ cd mystuff
# ... 使用 gedit 编辑 text.txt ...
[mystuff]$ ls
test.txt
[mystuff]$
```

你看到的命令行信息，Python 信息，以及其它一些东西可能会非常不一样。不过应该大致不差就是了。

给新手的告诫

你已经完成了这节练习，取决于你对计算机的熟悉程度，这个练习对你而言可能会有些难。如果你觉得有难度的话，你要自己克服困难，多花点时间学习一下。因为如果你不会这些基础操作的话，编程对你来说将会更难学习。

如果有程序员告诉你让你使用 `vim` 或者 `emacs`，那你应该拒绝他们。当你成为一个更好的程序员的时候，这些编辑器才会适合你使用。你现在需要的只是一个可以编辑文字的编辑器。我们使用 `gedit` 是因为它很简单，而且在不同的系统上面使用起来是一样的。就连专业程序员也会使用 `gedit`，所以对于初学而言它已经足够了。

也许有程序员会告诉你让你安装和学习 Python 3。你应该告诉他们"等你电脑里的所有 python 代码都支持 Python 3 了，我再试着学学吧。"你这句话足够他们忙活个十来年的了。

总有一天你会听到有程序员建议你使用 Mac OSX 或者 Linux。如果他喜欢字体美观，他会告诉你让你弄台 Mac OSX 计算机，如果他们喜欢操作控制而且留了一部大胡子，他会让你安装 Linux。这里再次向你说明，只要是一台手上能用的电脑就可以了。你需要的只有三样东西: `gedit`、一个命令行终端、还有 `python`。

最后要说的是这节练习的准备工作的目的，也就是让你可以在以后的练习中顺利地做到下面的这些事情：

1. 使用 `gedit` 编写代码。
2. 运行你写的习题。
3. 修改错误的习题。
4. 重复上述步骤。

其他的事情只会让你更困惑，所以还是坚持按计划进行吧。

习题 1：第一个程序

你应该在练习 0 中花了不少的时间，学会了如何安装文本编辑器、运行文本编辑器、以及如何运行命令行终端，而且你已经花时间熟悉了这些工具。请不要跳过前一个练习的内容直接进行下面的内容，这也是本书唯一的一次这样的警示。

```
print "Hello World!"
print "Hello Again"
print "I like typing this."
print "This is fun."
print 'Yay! Printing.'
print "I'd much rather you 'not'."
print 'I "said" do not touch this.'
```

将上面的内容写到一个文件中，取名为 `ex1.py`。注意这个命名方式，Python 文件最好以 `.py` 结尾。

> **Warning:** 不要把上面内容最左边的数字也输进去。这些是所谓的“行号 (line numbers)”，程序员在谈论到程序中某个位置的错误时会使用到行号。Python 在程序出错时也会以行号的方式告诉你错误信息，但是你是不需要输入这些行号的。

然后你需要在命令行终端通过输入以下内容来运行这段代码：

```
python ex1.py
```

如果你写对了的话，你应该看到和下面一样的内容。如果不一样，那就是你弄错了什么东西。不是计算机出错了，计算机没错。

你应该看到的内容

```
$ python ex1.py
Hello World!
Hello Again
I like typing this.
This is fun.
Yay! Printing.
I'd much rather you 'not'.
```

```
I "said" do not touch this.
$
```

你也许会看到 $ 前面会显示你所在的目录的名字，这不是问题，但如果你的输出不一样的话，你需要找出为什么会不一样，然后把你的程序改对。

如果你看到类似如下的错误信息：

```
$ python ex/ex1.py
  File "ex/ex1.py", line 3
    print "I like typing this.
                             ^
SyntaxError: EOL while scanning string literal
```

这些内容你应该学会看懂的，这是很重要的一点，因为你以后还会犯类似的错误。就是我现在也会犯这样的错误。让我们一行一行来看。

1. 首先我们在命令行终端输入命令来运行 `ex1.py` 脚本。
2. Python 告诉我们 `ex1.py` 文件的第 3 行有一个错误。
3. 然后这一行的内容被打印了出来。
4. 然后 Python 打印出一个 ^ (井号，caret) 符号，用来指示出错的位置。注意到少了一个 " (双引号，double-quote) 符号了吗？
5. 最后，它打印出了一个"语法错误 (SyntaxError)"告诉你究竟是什么样的错误。通常这些错误信息都非常难懂，不过你可以把错误信息的内容复制到搜索引擎里，然后你就能看到别人也遇到过这样的错误，而且你也许能找到如何解决这个问题。

> **Warning:** 如果你来自另外一个国家，而且你看到关于 ASCII 编码的错误，那就在你的 python 脚本的最上面加入这一行：
>
> *# -- coding: utf-8 --*
>
> 这样你就在脚本中使用了 unicode UTF-8 编码，这些错误就不会出现了。

加分习题

你还会有 加分习题 需要完成。加分习题里边的内容是供你尝试的。如果你觉得做不出来，你可以暂时跳过，过段时间再回来做。

在这个练习中，试试这些东西：

1. 让你的脚本再多打印一行。
2. 让你的脚本只打印一行。
3. 在一行的起始位置放一个 '#' (octothorpe) 符号。它的作用是什么？自己研究一下。

从现在开始，除非特别情况，我将不再解释每个习题的工作原理了。

Note: 井号有很多的英文名字，例如：'octothorpe(八角帽)'，'pound(英镑符)', 'hash(电话的 # 键)', 'mesh(网)' 等。

习题 1: 第一个程序

习题 2：注释和井号

程序里的注释是很重要的。它们可以用自然语言告诉你某段代码的功能是什么。在你想要临时移除一段代码时，你还可以用注解的方式将这段代码临时禁用。接下来的练习将让你学会注释:

```
# A comment, this is so you can read your program later.
# Anything after the # is ignored by python.

print "I could have code like this." # and the comment after is ignored

# You can also use a comment to "disable" or comment out a piece of code:
# print "This won't run."

print "This will run."
```

你应该看到的结果

```
$ python ex2.py
I could have code like this.
This will run.
$
```

加分习题

1. 弄清楚”#” 符号的作用。而且记住它的名字。(中文为井号，英文为 octothorpe 或者 pound character)。
2. 打开你的 `ex2.py` 文件，从后往前逐行检查。从最后一行开始，倒着逐个单词单词检查回去。
3. 有没有发现什么错误呢？有的话就改正过来.
4. 朗读你写的习题，把每个字符都读出来。有没有发现更多的错误呢？有的话也一样改正过来。

习题 3：数字和数学计算

每一种编程语言都包含处理数字和进行数学计算的方法。不必担心，程序员经常撒谎说他们是多么牛的数学天才，其实他们根本不是。如果他们真是数学天才，他们早就去从事数学相关的行业了，而不是写写广告程序和社交网络游戏，从人们身上偷赚点小钱而已。

这章练习里有很多的数学运算符号。我们来看一遍它们都叫什么名字。你要一边写一边念出它们的名字来，直到你念烦了为止。名字如下：

- + plus 加号
- - minus 减号
- / slash 斜杠
- * asterisk 星号
- % percent 百分号
- < less-than 小于号
- > greater-than 大于号
- <= less-than-equal 小于等于号
- >= greater-than-equal 大于等于号

有没有注意到以上只是些符号，没有运算操作呢？写完下面的练习代码后，再回到上面的列表，写出每个符号的作用。例如 + 是用来做加法运算的。

```
print "I will now count my chickens:"

print "Hens", 25 + 30 / 6
print "Roosters", 100 - 25 * 3 % 4

print "Now I will count the eggs:"

print 3 + 2 + 1 - 5 + 4 % 2 - 1 / 4 + 6

print "Is it true that 3 + 2 < 5 - 7?"

print 3 + 2 < 5 - 7
```

```python

print "What is 3 + 2?", 3 + 2
print "What is 5 - 7?", 5 - 7

print "Oh, that's why it's False."

print "How about some more."

print "Is it greater?", 5 > -2
print "Is it greater or equal?", 5 >= -2
print "Is it less or equal?", 5 <= -2
```

你应该看到的结果

```
$ python ex3.py
I will now count my chickens:
Hens 30
Roosters 97
Now I will count the eggs:
7
Is it true that 3 + 2 < 5 - 7?
False
What is 3 + 2? 5
What is 5 - 7? -2
Oh, that's why it's False.
How about some more.
Is it greater? True
Is it greater or equal? True
Is it less or equal? False
$
```

加分习题

1. 使用 `#` 在代码每一行的前一行为自己写一个注解，说明一下这一行的作用。
2. 记得开始时的 < 练习 0> 吧？用里边的方法把 Python 运行起来，然后使用刚才学到的运算符号，把 Python 当做计算器玩玩。
3. 自己找个想要计算的东西，写一个 `.py` 文件把它计算出来。
4. 有没有发现计算结果是”错”的呢？计算结果只有整数，没有小数部分。研究一下这是为什么，搜索一下“浮点数 (floating point number)”是什么东西。
5. 使用浮点数重写一遍 `ex3.py`，让它的计算结果更准确 (提示: 20.0 是一个浮点数)。

习题 4: 变量 (variable) 和命名

你已经学会了 print 和算术运算。下一步你要学的是“变量”。在编程中，变量只不过是用来指代某个东西的名字。程序员通过使用变量名可以让他们的程序读起来更像英语。而且因为程序员的记性都不怎么地，变量名可以让他们更容易记住程序的内容。如果他们没有在写程序时使用好的变量名，在下一次读到原来写的代码时他们会大为头疼的。

如果你被这章习题难住了的话，记得我们之前教过的：找到不同点、注意细节。

1. 在每一行的上面写一行注解，给自己解释一下这一行的作用。
2. 倒着读你的 .py 文件。
3. 朗读你的 .py 文件，将每个字符也朗读出来。

```
cars = 100
space_in_a_car = 4.0
drivers = 30
passengers = 90
cars_not_driven = cars - drivers
cars_driven = drivers
carpool_capacity = cars_driven * space_in_a_car
average_passengers_per_car = passengers / cars_driven


print "There are", cars, "cars available."
print "There are only", drivers, "drivers available."
print "There will be", cars_not_driven, "empty cars today."
print "We can transport", carpool_capacity, "people today."
print "We have", passengers, "to carpool today."
print "We need to put about", average_passengers_per_car, "in each car."
```

Note: space_in_a_car 中的 _ 是 下划线 (underscore) 字符。你要自己学会怎样打出这个字符来。这个符号在变量里通常被用作假想的空格，用来隔开单词。

你应该看到的结果

```
$ python ex4.py
There are 100 cars available.
There are only 30 drivers available.
There will be 70 empty cars today.
We can transport 120.0 people today.
We have 90 to carpool today.
We need to put about 3 in each car.
$
```

加分习题

当我刚开始写这个程序时我犯了个错误，python 告诉我这样的错误信息：

```
Traceback (most recent call last):
  File "ex4.py", line 8, in <module>
    average_passengers_per_car = car_pool_capacity / passenger
NameError: name 'car_pool_capacity' is not defined
```

用你自己的话解释一下这个错误信息，解释时记得使用行号，而且要说明原因。

更多的加分习题:

1. 我在程序里用了 4.0 作为 `space_in_a_car` 的值，这样做有必要吗？如果只用 4 会有什么问题?
2. 记住 4.0 是一个“浮点数”，自己研究一下这是什么意思。
3. 在每一个变量赋值的上一行加上一行注解。
4. 记住 `=` 的名字是等于 (equal)，它的作用是为东西取名。
5. 记住 `_` 是下划线字符 (underscore)。
6. 将 `python` 作为计算器运行起来，就跟以前一样，不过这一次在计算过程中使用变量名来做计算，常见的变量名有 `i`, `x`, `j` 等等。

习题 5：更多的变量和打印

我们现在要键入更多的变量并且把它们打印出来。这次我们将使用一个叫“格式化字符串 (format string)”的东西. 每一次你使用 " 把一些文本引用起来，你就建立了一个字符串。字符串是程序将信息展示给人的方式。你可以打印它们，可以将它们写入文件，还可以将它们发送给网站服务器，很多事情都是通过字符串交流实现的。

字符串是非常好用的东西，所以再这个练习中你将学会如何创建包含变量内容的字符串。使用专门的格式和语法把变量的内容放到字符串里，相当于来告诉 python ：“嘿，这是一个格式化字符串，把这些变量放到那几个位置。”

一样的，即使你读不懂这些内容，只要一字不差地键入就可以了。

```
my_name = 'Zed A. Shaw'
my_age = 35 # not a lie
my_height = 74 # inches
my_weight = 180 # lbs
my_eyes = 'Blue'
my_teeth = 'White'
my_hair = 'Brown'

print "Let's talk about %s." % my_name
print "He's %d inches tall." % my_height
print "He's %d pounds heavy." % my_weight
print "Actually that's not too heavy."
print "He's got %s eyes and %s hair." % (my_eyes, my_hair)
print "His teeth are usually %s depending on the coffee." % my_teeth

# this line is tricky, try to get it exactly right
print "If I add %d, %d, and %d I get %d." % (
    my_age, my_height, my_weight, my_age + my_height + my_weight)
```

Warning: 如果你使用了非 ASCII 字符而且碰到了编码错误，记得在最顶端加一行 `# -- coding: utf-8 --` 。

你应该看到的结果

```
$ python ex5.py
Let's talk about Zed A. Shaw.
He's 74 inches tall.
He's 180 pounds heavy.
Actually that's not too heavy.
He's got Blue eyes and Brown hair.
His teeth are usually White depending on the coffee.
If I add 35, 74, and 180 I get 289.
$
```

加分习题

1. 修改所有的变量名字，把它们前面的 ‘‘my_‘‘去掉。确认将每一个地方的都改掉，不只是你使用 ‘‘=‘‘赋值过的地方。
2. 试着使用更多的格式化字符。例如 `%r` 就是是非常有用的一个，它的含义是“不管什么都打印出来”。
3. 在网上搜索所有的 Python 格式化字符。
4. 试着使用变量将英寸和磅转换成厘米和千克。不要直接键入答案。使用 Python 的计算功能来完成。

习题 6：字符串 (string) 和文本

虽然你已经在程序中写过字符串了，你还没学过它们的用处。在这章习题中我们将使用复杂的字符串来建立一系列的变量，从中你将学到它们的用途。首先我们解释一下字符串是什么东西。

字符串通常是指你想要展示给别人的、或者是你想要从程序里“导出”的一小段字符。Python 可以通过文本里的双引号 `"` 或者单引号 `'` 识别出字符串来。这在你以前的 `print` 练习中你已经见过很多次了。如果你把单引号或者双引号括起来的文本放到 `print` 后面，它们就会被 python 打印出来。

字符串可以包含格式化字符 `%s`，这个你之前也见过的。你只要将格式化的变量放到字符串中，再紧跟着一个百分号 `%` (percent)，再紧跟着变量名即可。唯一要注意的地方，是如果你想要在字符串中通过格式化字符放入多个变量的时候，你需要将变量放到 `( )` 圆括号 (parenthesis) 中，而且变量之间用 `,` 逗号 (comma) 隔开。就像你逛商店说“我要买牛奶、面包、鸡蛋、八宝粥”一样，只不过程序员说的是”(milk, eggs, bread, soup)”。

我们将键入大量的字符串、变量、和格式化字符，并且将它们打印出来。我们还将练习使用简写的变量名。程序员喜欢使用恼人的难度的简写来节约打字时间，所以我们现在就提早学会这个，这样你就能读懂并且写出这些东西了。

```python
x = "There are %d types of people." % 10
binary = "binary"
do_not = "don't"
y = "Those who know %s and those who %s." % (binary, do_not)

print x
print y

print "I said: %r." % x
print "I also said: '%s'." % y

hilarious = False
joke_evaluation = "Isn't that joke so funny?! %r"

print joke_evaluation % hilarious

w = "This is the left side of..."
e = "a string with a right side."
```

```python

print w + e
```

你应该看到的结果

```
$ python ex6.py
There are 10 types of people.
Those who know binary and those who don't.
I said: 'There are 10 types of people.'.
I also said: 'Those who know binary and those who don't.'.
Isn't that joke so funny?! False
This is the left side of...a string with a right side.
$
```

加分习题

1. 通读程序，在每一行的上面写一行注解，给自己解释一下这一行的作用。
2. 找到所有的”字符串包含字符串”的位置，总共有四个位置。
3. 你确定只有四个位置吗？你怎么知道的？没准我在骗你呢。
4. 解释一下为什么 `w` 和 `e` 用 `+` 连起来就可以生成一个更长的字符串。

习题 7：更多打印

现在我们将做一批练习，在练习的过程中你需要键入代码，并且让它们运行起来。我不会解释太多，因为这节的内容都是以前熟悉过的。这节练习的目的是巩固你学到的东西。我们几个练习后再见。不要跳过这些习题。不要复制粘贴！

```
print "Mary had a little lamb."
print "Its fleece was white as %s." % 'snow'
print "And everywhere that Mary went."
print "." * 10  # what'd that do?

end1 = "C"
end2 = "h"
end3 = "e"
end4 = "e"
end5 = "s"
end6 = "e"
end7 = "B"
end8 = "u"
end9 = "r"
end10 = "g"
end11 = "e"
end12 = "r"

# watch that comma at the end.  try removing it to see what happens
print end1 + end2 + end3 + end4 + end5 + end6,
print end7 + end8 + end9 + end10 + end11 + end12
```

你应该看到的结果

```
$ python ex7.py
Mary had a little lamb.
Its fleece was white as snow.
And everywhere that Mary went.
..........
```

```
Cheese Burger
$
```

加分习题

接下来几节的加分习题是一样的。

1. 逆向阅读，在每一行的上面加一行注解。
2. 倒着朗读出来，找出自己的错误。
3. 从现在开始，把你的错误记录下来，写在一张纸上。
4. 在开始下一节习题时，阅读一遍你记录下来的错误，并且尽量避免在下个练习中再犯同样的错误。
5. 记住，每个人都会犯错误。程序员和魔术师一样，他们希望大家认为他们从不犯错，不过这只是表象而已，他们每时每刻都在犯错。

习题 8：打印，打印

```
formatter = "%r %r %r %r"

print formatter % (1, 2, 3, 4)
print formatter % ("one", "two", "three", "four")
print formatter % (True, False, False, True)
print formatter % (formatter, formatter, formatter, formatter)
print formatter % (
    "I had this thing.",
    "That you could type up right.",
    "But it didn't sing.",
    "So I said goodnight."
)
```

你应该看到的结果

```
$ python ex8.py
1 2 3 4
'one' 'two' 'three' 'four'
True False False True
'%r %r %r %r' '%r %r %r %r' '%r %r %r %r' '%r %r %r %r'
'I had this thing.' 'That you could type up right.' "But it didn't sing." 'So I said goodnight.'
$
```

加分习题

1. 自己检查结果，记录你犯过的错误，并且在下个练习中尽量不犯同样的错误。
2. 注意最后一行程序中既有单引号又有双引号，你觉得它是如何工作的？

习题 9：打印，打印，打印

```python
# Here's some new strange stuff, remember type it exactly.

days = "Mon Tue Wed Thu Fri Sat Sun"
months = "Jan\nFeb\nMar\nApr\nMay\nJun\nJul\nAug"

print "Here are the days: ", days
print "Here are the months: ", months

print """
There's something going on here.
With the three double-quotes.
We'll be able to type as much as we like.
Even 4 lines if we want, or 5, or 6.
"""
```

你应该看到的结果

```
$ python ex9.py
Here are the days:  Mon Tue Wed Thu Fri Sat Sun
Here are the months:  Jan
Feb
Mar
Apr
May
Jun
Jul
Aug

There's something going on here.
With the three double-quotes.
We'll be able to type as much as we like.
Even 4 lines if we want, or 5, or 6.
```

```
$
```

加分习题

1. 自己检查结果，记录你犯过的错误，并且在下个练习中尽量不犯同样的错误。

习题 10: 那是什么？

在习题 9 中我你接触了一些新东西。我让你看到两种让字符串扩展到多行的方法。第一种方法是在月份之间用 `\n` (back-slash `n`) 隔开。这两个字符的作用是在该位置上放入一个“新行 (new line)”字符。

使用反斜杠 \ (back-slash) 可以将难打印出来的字符放到字符串。针对不同的符号有很多这样的所谓“转义序列 (escape sequences)”，但有一个特殊的转义序列，就是 双反斜杠 `(double back-slash) \\` 。这两个字符组合会打印出一个反斜杠来。接下来我们做几个练习，然后你就知道这些转义序列的意义了。

另外一种重要的转义序列是用来将单引号 `'` 和双引号 `"` 转义。想象你有一个用双引号引用起来的字符串，你想要在字符串的内容里再添加一组双引号进去，比如你想说`"I "understand" joe."`，Python 就会认为 `"understand"` 前后的两个引号是字符串的边界，从而把字符串弄错。你需要一种方法告诉 python 字符串里边的双引号不是真正的双引号。

要解决这个问题，你需要将双引号和单引号转义，让 Python 将引号也包含到字符串里边去。这里有一个例子：

```
"I am 6'2\" tall."  # 将字符串中的双引号转义
'I am 6\'2" tall.'  # 将字符串种的单引号转义
```

第二种方法是使用“三引号 (triple-quotes)”，也就是 `"""`，你可以在一组三引号之间放入任意多行的文字。接下来你将看到用法。

```
tabby_cat = "\tI'm tabbed in."
persian_cat = "I'm split\non a line."
backslash_cat = "I'm \\ a \\ cat."

fat_cat = """
I'll do a list:
\t* Cat food
\t* Fishies
\t* Catnip\n\t* Grass
"""

print tabby_cat
print persian_cat
print backslash_cat
print fat_cat
```

你应该看到的结果

注意你打印出来的制表符，这节练习中的文字间隔对于答案的正确性是很重要的。

```
$ python ex10.py
        I'm tabbed in.
I'm split
on a line.
I'm \ a \ cat.

I'll do a list:
        * Cat food
        * Fishies
        * Catnip
        * Grass

$
```

加分习题

1. 上网搜索一下还有哪些可用的转义字符。
2. 使用 `'''` (三个单引号) 取代三个双引号，看看效果是不是一样的？
3. 将转义序列和格式化字符串放到一起，创建一种更复杂的格式。
4. 记得 `%r` 格式化字符串吗？使用 `%r` 搭配单引号和双引号转义字符打印一些字符串出来。将 %r 和 %s 比较一下。注意到了吗？%r 打印出来的是你写在脚本里的内容，而 %s 打印的是你应该看到的内容。

习题 11：提问

我已经出过很多打印相关的练习，让你习惯写简单的东西，但简单的东西都有点无聊，现在该跟上脚步了。我们现在要做的是把数据读到你的程序里边去。这可能对你有点难度，你可能一下子不明白，不过你需要相信我，无论如何把习题做了再说。只要做几个练习你就明白了。

一般软件做的事情主要就是下面几条：

1. 接受人的输入。
2. 改变输入。
3. 打印出改变了的输入。

到目前为止你只做了打印，但还不会接受或者修改人的输入。你也许还不知道“输入 (input)”是什么意思。所以闲话少说，我们还是开始做点练习看你能不能明白。下一个习题里边我们会给你更多的解释。

```
print "How old are you?",
age = raw_input()
print "How tall are you?",
height = raw_input()
print "How much do you weigh?",
weight = raw_input()

print "So, you're %r old, %r tall and %r heavy." % (
    age, height, weight)
```

Note: 注意到我在每行 print 后面加了个逗号 (comma) ，了吧？这样的话 print 就不会输出新行符而结束这一行跑到下一行去了。

你应该看到的结果

```
$ python ex11.py
How old are you? 35
How tall are you? 6'2"
How much do you weigh? 180lbs
```

```
So, you're '35' old, '6\'2"' tall and '180lbs' heavy.
$
```

加分习题

1. 上网查一下 Python 的 `raw_input` 实现的是什么功能。
2. 你能找到它的别的用法吗？测试一下你上网搜索到的例子。
3. 用类似的格式再写一段，把问题改成你自己的问题。
4. 和转义序列有关的，想想为什么最后一行 `'6\'2"'` 里边有一个 `\'` 序列。单引号需要被转义，从而防止它被识别为字符串的结尾。有没有注意到这一点？

习题 12：提示别人

当你键入 `raw_input()` 的时候，你需要键入 (和) 也就是“括号 (parenthesis)”。这和你格式化输出两个以上变量时的情况有点类似，比如说 `"%s %s" % (x, y)` 里边就有括号。对于 `raw_input` 而言，你还可以让它显示出一个提示，从而告诉别人应该输入什么东西。你可以在 () 之间放入一个你想要作为提示的字符串，如下所示：

```
y = raw_input("Name? ")
```

这句话会用“Name?”提示用户，然后将用户输入的结果赋值给变量 y。这就是我们提问用户并且得到答案的方式。

也就是说，我们的上一个练习可以使用 `raw_input` 重写一次。所有的提示都可以通过 `raw_input` 实现。

```
age = raw_input("How old are you? ")
height = raw_input("How tall are you? ")
weight = raw_input("How much do you weigh? ")

print "So, you're %r old, %r tall and %r heavy." % (
    age, height, weight)
```

你应该看到的结果

```
$ python ex12.py
How old are you? 35
How tall are you? 6'2"
How much do you weight? 180lbs
So, you're '35' old, '6\'2"' tall and '180lbs' heavy.
$
```

加分习题

1. 在命令行界面下运行你的程序，然后在命令行输入 `pydoc raw_input` 看它说了些什么。如果你用的是 Window，那就试一下 `python -m pydoc raw_input` 。

2. 输入 `q` 退出 pydoc。
3. 上网找一下 `pydoc` 命令是用来做什么的。
4. 使用 pydoc 再看一下 `open`, `file`, `os`, 和 `sys` 的含义。看不懂没关系，只要通读一下，记下你觉得有意思的点就行了。

习题 13：参数、解包、变量

在这节练习中，我们将降到另外一种将变量传递给脚本的方法 (所谓脚本，就是你写的 .py 程序)。你已经知道，如果要运行 ex13.py，只要在命令行运行 python ex13.py 就可以了。这句命令中的 ex13.py 部分就是所谓的“参数 (argument)”，我们现在要做的就是写一个可以接受参数的脚本。

将下面的程序写下来，后面你将看到详细解释。

```
1 from sys import argv
2 
3 script, first, second, third = argv
4 
5 print "The script is called:", script
6 print "Your first variable is:", first
7 print "Your second variable is:", second
8 print "Your third variable is:", third
```

在第 1 行我们有一个“import”语句. 这是你将 python 的功能引入你的脚本的方法. Python 不会一下子将它所有的功能给你，而是让你需要什么就调用什么。这样可以让你的程序保持精简，而后面的程序员看到你的代码的时候，这些“import”可以作为提示，让他们明白你的代码用到了哪些功能。

argv 是所谓的“参数变量 (argument variable)”，是一个非常标准的编程术语。在其他的编程语言里你也可以看到它。这个变量包含了你传递给 Python 的参数。通过后面的练习你将对它有更多的了解。

第 3 行将 argv “解包 (unpack)”，与其将所有参数放到同一个变量下面，我们将每个参数赋予一个变量名：script, first, second, 以及 third。这也许看上去有些奇怪, 不过”解包”可能是最好的描述方式了。它的含义很简单：“把 argv 中的东西解包，将所有的参数依次赋予左边的变量名”。

接下来就是正常的打印了。

等一下！“功能”还有另外一个名字

前面我们使用 import 让你的程序实现更多的功能，但实际上没人吧 import 称为“功能”。我希望你可以在没接触到正式术语的时候就弄懂它的功能。在继续下去之前, 你需要知道它们的真正名称：模组 (modules)。

从现在开始我们将把这些我们导入 (`import`)进来的功能称作模组。你将看到类似这样的说法：“你需要把 `sys` 模组 import 进来。”也有人将它们称作“库 (libraries)”，不过我们还是叫它们模组吧。

你应该看到的结果

用下面的方法运行你的程序（注意你必须传递 * 三 * 个参数）：

```
python ex13.py first 2nd 3rd
```

如果你每次使用不同的参数运行，你将看到下面的结果：

```
$ python ex13.py first 2nd 3rd
The script is called: ex13.py
Your first variable is: first
Your second variable is: 2nd
Your third variable is: 3rd

$ python ex13.py cheese apples bread
The script is called: ex13.py
Your first variable is: cheese
Your second variable is: apples
Your third variable is: bread

$ python ex13.py Zed A. Shaw
The script is called: ex13.py
Your first variable is: Zed
Your second variable is: A.
Your third variable is: Shaw
```

你其实可以将“first”、“2nd”、“3rd”替换成任意三样东西。你可以将它们换成任意你想要的东西.

```
python ex13.py stuff I like
python ex13.py anything 6 7
```

如果你没有运行对，你将看到如下错误：

```
python ex13.py first 2nd
Traceback (most recent call last):
  File "ex/ex13.py", line 3, in <module>
    script, first, second, third = argv
ValueError: need more than 3 values to unpack
```

当你运行脚本时提供的参数的个数不对的时候，你就会看到上述错误信息 (这次我只用了 `first 2nd` 两个
参数)。“need more than 3 values to unpack”这个错误信息告诉你参数数量不足。

加分习题

1. 给你的脚本三个以下的参数。看看会得到什么错误信息。试着解释一下。
2. 再写两个脚本，其中一个接受更少的参数，另一个接受更多的参数，在参数解包时给它们取一些有意义的变量名。
3. 将 `raw_input` 和 `argv` 一起使用，让你的脚本从用户手上得到更多的输入。
4. 记住“模组 (modules)”为你提供额外功能。多读几遍把这个词记住，因为我们后面还会用到它。

习题 14：提示和传递

让我们使用 argv 和 raw_input 一起来向用户提一些特别的问题。下一节习题你会学习如何读写文件，这节练习是下节的基础。在这道习题里我们将用略微不同的方法使用 raw_input，让它打出一个简单的 > 作为提示符。这和一些游戏中的方式类似，例如 Zork 或者 Adventure 这两款游戏。

```
from sys import argv

script, user_name = argv
prompt = '> '

print "Hi %s, I'm the %s script." % (user_name, script)
print "I'd like to ask you a few questions."
print "Do you like me %s?" % user_name
likes = raw_input(prompt)

print "Where do you live %s?" % user_name
lives = raw_input(prompt)

print "What kind of computer do you have?"
computer = raw_input(prompt)

print """
Alright, so you said %r about liking me.
You live in %r.  Not sure where that is.
And you have a %r computer.  Nice.
""" % (likes, lives, computer)
```

我们将用户提示符设置为变量 prompt，这样我们就不需要在每次用到 raw_input 时重复输入提示用户的字符了。而且如果你要将提示符修改成别的字串，你只要改一个位置就可以了。

非常顺手吧。

你应该看到的结果

当你运行这个脚本时，记住你需要把你的名字赋给这个脚本，让 argv 参数接收到你的名称。

```
$ python ex14.py Zed
Hi Zed, I'm the ex14.py script.
I'd like to ask you a few questions.
Do you like me Zed?
> yes
Where do you live Zed?
> America
What kind of computer do you have?
> Tandy

Alright, so you said 'yes' about liking me.
You live in 'America'.  Not sure where that is.
And you have a 'Tandy' computer.  Nice.
```

加分习题

1. 查一下 Zork 和 Adventure 是两个怎样的游戏。看看能不能下载到一版，然后玩玩看。
2. 将 `prompt` 变量改成完全不同的内容再运行一遍。
3. 给你的脚本再添加一个参数，让你的程序用到这个参数。
4. 确认你弄懂了三个引号 `"""` 可以定义多行字符串，而 `%` 是字符串的格式化工具。

习题 15：读取文件

你已经学过了 raw_input 和 argv，这些是你开始学习读取文件的必备基础。你可能需要多多实验才能明白它的工作原理，所以你要细心做练习，并且仔细检查结果。处理文件需要非常仔细，如果不仔细的话，你可能会吧有用的文件弄坏或者清空。导致前功尽弃。

这节练习涉及到写两个文件。一个正常的 ex15.py 文件，另外一个是 ex15_sample.txt，第二个文件并不是脚本，而是供你的脚本读取的文本文件。以下是后者的内容：

```
This is stuff I typed into a file.
It is really cool stuff.
Lots and lots of fun to have in here.
```

我们要做的是把该文件用我们的脚本"打开 (open)"，然后打印出来。然而把文件名 ex15_sample.txt 写死 (hardcode) 在代码中不是一个好主意，这些信息应该是用户输入的才对。如果我们碰到其他文件要处理，写死的文件名就会给你带来麻烦了。我们的解决方案是使用 argv 和 raw_input 来从用户获取信息，从而知道哪些文件该被处理。

```
1  from sys import argv
2
3  script, filename = argv
4
5  txt = open(filename)
6
7  print "Here's your file %r:" % filename
8  print txt.read()
9
10 print "Type the filename again:"
11 file_again = raw_input("> ")
12
13 txt_again = open(file_again)
14
15 print txt_again.read()
```

这个脚本中有一些新奇的玩意，我们来快速地过一遍：

代码的 1-3 行使用 argv 来获取文件名，这个你应该已经熟悉了。接下来第 5 行我们看到 open 这个新命令。现在请在命令行运行 pydoc open 来读读它的说明。你可以看到它和你自己的脚本、或者 raw_input

命令类似，它会接受一个参数，并且返回一个值，你可以将这个值赋予一个变量。这就是你打开文件的过程。

第 7 行我们打印了一小行，但在第 8 行我们看到了新奇的东西。我们在 `txt` 上调用了一个函数。你从 open 获得的东西是一个 `file` (文件)，文件本身也支持一些命令。它接受命令的方式是使用句点 `.` (英文称作 dot 或者 period)，紧跟着你的命令，然后是类似 `open` 和 `raw_input` 一样的参数。不同点是：当你说 `txt.read` 时，你的意思其实是："嘿 txt！执行你的 read 命令，无需任何参数！"

脚本剩下的部分基本差不多，不过我就把剩下的分析作为加分习题留给你自己了。

你应该看到的结果

我的脚本叫 "ex15_sample.txt"，以下是执行结果：

```
$ python ex15.py ex15_sample.txt
Here's your file 'ex15_sample.txt':
This is stuff I typed into a file.
It is really cool stuff.
Lots and lots of fun to have in here.


Type the filename again:
> ex15_sample.txt
This is stuff I typed into a file.
It is really cool stuff.
Lots and lots of fun to have in here.


$
```

加分习题

这节的难度跨越有点大，所以你要尽量做好这节加分习题，然后再继续后面的章节。

1. 在每一行的上面用注解说明这一行的用途。
2. 如果你不确定答案，就问别人，或者上网搜索。大部分时候，只要搜索 "python" 加上你要搜的东西就能得到你要的答案。比如搜索一下 "python open"。
3. 我使用了 "命令" 这个词，不过实际上它们的名字是 "函数（function）" 和 "方法（method）。上网搜索一下这两者的意义和区别。看不明白也没关系，迷失在别的程序员的知识海洋里是很正常的一件事情。
4. 删掉 10-15 行使用到 `raw_input` 的部分，再运行一遍脚本。
5. 只是用 `raw_input` 写这个脚本，想想那种得到文件名称的方法更好，以及为什么。

6. 运行 `pydoc file` 向下滚动直到看见 `read()` 命令（函数/方法）。看到很多别的命令了吧，你可以找几条试试看。不需要看那些包含 `__` （两个下划线）的命令，这些只是垃圾而已。
7. 再次运行 `python` 在命令行下使用 `open` 打开一个文件，这种 open 和 read 的方法也值得你一学。
8. 让你的脚本针对 `txt` and `txt_again` 变量执行一下 `close()` ，处理完文件后你需要将其关闭，这是很重要的一点。

习题 16: 读写文件

如果你做了上一个练习的加分习题，你应该已经了解了各种文件相关的命令（方法/函数）。你应该记住的命令如下：

- close – 关闭文件。跟你编辑器的 `文件 -> 保存..` 一个意思。
- read – 读取文件内容。你可以把结果赋给一个变量。
- readline – 读取文本文件中的一行。
- truncate – 清空文件，请小心使用该命令。
- write(stuff) – 将 stuff 写入文件。

这是你现在该知道的重要命令。有些命令需要接受参数，这对我们并不重要。你只要记住 `write` 的用法就可以了。`write` 需要接收一个字符串作为参数，从而将该字符串写入文件。

让我们来使用这些命令做一个简单的文本编辑器吧:

```
from sys import argv

script, filename = argv

print "We're going to erase %r." % filename
print "If you don't want that, hit CTRL-C (^C)."
print "If you do want that, hit RETURN."

raw_input("?")

print "Opening the file..."
target = open(filename, 'w')

print "Truncating the file.  Goodbye!"
target.truncate()

print "Now I'm going to ask you for three lines."

line1 = raw_input("line 1: ")
line2 = raw_input("line 2: ")
```

```
21 line3 = raw_input("line 3: ")
22
23 print "I'm going to write these to the file."
24
25 target.write(line1)
26 target.write("\n")
27 target.write(line2)
28 target.write("\n")
29 target.write(line3)
30 target.write("\n")
31
32 print "And finally, we close it."
33 target.close()
```

这个文件是够大的，大概是你键入过的最大的文件。所以慢慢来，仔细检查，让它能运行起来。有一个小技巧就是你可以让你的脚本一部分一部分地运行起来。先写 1-8 行，让它运行起来，再多运行 5 行，再接着多运行几行，以此类推，直到整个脚本运行起来为止。

你应该看到的结果

你将看到两样东西，一样是你新脚本的输出:

```
$ python ex16.py test.txt
We're going to erase 'test.txt'.
If you don't want that, hit CTRL-C (^C).
If you do want that, hit RETURN.
?
Opening the file...
Truncating the file.  Goodbye!
Now I'm going to ask you for three lines.
line 1: To all the people out there.
line 2: I say I don't like my hair.
line 3: I need to shave it off.
I'm going to write these to the file.
And finally, we close it.
$
```

接下来打开你新建的文件（我的是 `test.txt` ）检查一下里边的内容，怎么样，不错吧？

加分习题

1. 如果你觉得自己没有弄懂的话，用我们的老办法，在每一行之前加上注解，为自己理清思路。就算不能理清思路，你也可以知道自己究竟具体哪里没弄明白。

2. 写一个和上一个练习类似的脚本，使用 `read` 和 `argv` 读取你刚才新建的文件。
3. 文件中重复的地方太多了。试着用一个 `target.write()` 将 `line1`, `line2`, `line3` 打印出来，你可以使用字符串、格式化字符、以及转义字符。
4. 找出为什么我们需要给 `open` 多赋予一个 `'w'` 参数。提示：`open` 对于文件的写入操作态度是安全第一，所以你只有特别指定以后，它才会进行写入操作。

习题 17：更多文件操作

现在让我们再学习几种文件操作。我们将编写一个 Python 脚本，将一个文件中的内容拷贝到另外一个文件中。这个脚本很短，不过它会让你对于文件操作有更多的了解。

```
from sys import argv
from os.path import exists

script, from_file, to_file = argv

print "Copying from %s to %s" % (from_file, to_file)

# we could do these two on one line too, how?
input = open(from_file)
indata = input.read()

print "The input file is %d bytes long" % len(indata)

print "Does the output file exist? %r" % exists(to_file)
print "Ready, hit RETURN to continue, CTRL-C to abort."
raw_input()

output = open(to_file, 'w')
output.write(indata)

print "Alright, all done."

output.close()
input.close()
```

你应该很快注意到了我们 `import` 了又一个很好用的命令 `exists`。这个命令将文件名字符串作为参数，如果文件存在的话，它将返回 `True`，否则将返回 `False`。在本书的下半部分，我们将使用这个函数做很多的事情，不过现在你应该学会怎样通过 `import` 调用它。

通过使用 `import` ，你可以在自己代码中直接使用其他更厉害的（通常是这样，不过也不尽然）程序员写的大量免费代码，这样你就不需要重写一遍了。

你应该看到的结果

和你前面写的脚本一样，运行该脚本需要两个参数，一个是待拷贝的文件，一个是要拷贝至的文件。如果我们使用以前的 test.txt 我们将看到如下的结果:

```
$ python ex17.py test.txt copied.txt
Copying from test.txt to copied.txt
The input file is 81 bytes long
Does the output file exist? False
Ready, hit RETURN to continue, CTRL-C to abort.

Alright, all done.

$ cat copied.txt
To all the people out there.
I say I don't like my hair.
I need to shave it off.
$
```

该命令对于任何文件都应该是有效的。试试操作一些别的文件看看结果。不过小心别把你的重要文件给弄坏了。

> **Warning:** 你看到我用 cat 这个命令了吧？它只能在 Linux 和 OSX 下面使用，使用 Windows 的就只好跟你说声抱歉了。

加分习题

1. 再多读读和 import 相关的材料，将 python 运行起来，试试这一条命令。试着看看自己能不能摸出点门道，当然了，即使弄不明白也没关系。
2. 这个脚本 实在是 有点烦人。没必要在拷贝之前问一遍把，没必要在屏幕上输出那么多东西。试着删掉脚本的一些功能，让它使用起来更加友好。
3. 看看你能把这个脚本改多短，我可以把它写成一行。
4. 我使用了一个叫 *cat* 的东西，这个古老的命令的用处是将两个文件“连接 (con*cat*enate)”到一起，不过实际上它最大的用途是打印文件内容到屏幕上。你可以通过 man cat 命令了解到更多信息。
5. 使用 Windows 的同学，你们可以给自己找一个 cat 的替代品。关于 man 的东西就别想太多了，Windows 下没这个命令。
6. 找出为什么你需要在代码中写 output.close() 。

习题 18：命名、变量、代码、函数

标题包含的内容够多的吧？接下来我要教你“函数 (function)”了！咚咚锵！说到函数，不一样的人会对它有不一样的理解和使用方法，不过我只会教你现在能用到的最简单的使用方式。

函数可以做三样事情：

1. 它们给代码片段命名，就跟“变量”给字符串和数字命名一样。
2. 它们可以接受参数，就跟你的脚本接受 `argv` 一样。
3. 通过使用 #1 和 #2，它们可以让你创建“微型脚本”或者“小命令”。

你可以使用 `def` 新建函数。我将让你创建四个不同的函数，它们工作起来和你的脚本一样。然后我会演示给你各个函数之间的关系。

```
# this one is like your scripts with argv
def print_two(*args):
    arg1, arg2 = args
    print "arg1: %r, arg2: %r" % (arg1, arg2)

# ok, that *args is actually pointless, we can just do this
def print_two_again(arg1, arg2):
    print "arg1: %r, arg2: %r" % (arg1, arg2)

# this just takes one argument
def print_one(arg1):
    print "arg1: %r" % arg1

# this one takes no arguments
def print_none():
    print "I got nothin'."


print_two("Zed","Shaw")
print_two_again("Zed","Shaw")
print_one("First!")
print_none()
```

让我们把你一个函数 `print_two` 肢解一下，这个函数和你写脚本的方式差不多，因此你看上去应该会觉着比较眼熟：

1. 首先我们告诉 Python 创建一个函数，我们使用到的命令是 `def` ，也就是“定义 (define)”的意思。
2. 紧接着 `def` 的是函数的名称。本例中它的名称是 “print_two”，但名字可以随便取，就叫 “peanuts” 也没关系。但最好函数的名称能够体现出函数的功能来。
3. 然后我们告诉函数我们需要 `*args` (asterisk args)，这和脚本的 `argv` 非常相似，参数必须放在圆括号 `()` 中才能正常工作。
4. 接着我们用冒号 `:` 结束本行，然后开始下一行缩进。
5. 冒号以下，使用 4 个空格缩进的行都是属于 `print_two` 这个函数的内容。其中第一行的作用是将参数解包，这和脚本参数解包的原理差不多。
6. 为了演示它的工作原理，我们把解包后的每个参数都打印出来，这和我们在之前脚本练习中所作的类似。

函数 `print_two` 的问题是：它并不是创建函数最简单的方法。在 Python 函数中我们可以跳过整个参数解包的过程，直接使用 `()` 里边的名称作为变量名。这就是 `print_two_again` 实现的功能。

接下来的例子是 `print_one` ，它向你演示了函数如何接受单个参数。

最后一个例子是 `print_none` ，它向你演示了函数可以不接收任何参数。

> **Warning:** 如果你不太能看懂上面的内容也别气馁。后面我们还有更多的练习向你展示如何创建和使用函数。现在你只要把函数理解成“迷你脚本”就可以了。

你应该看到的结果

运行上面的脚本会看到如下结果:

```
$ python ex18.py
arg1: 'Zed', arg2: 'Shaw'
arg1: 'Zed', arg2: 'Shaw'
arg1: 'First!'
I got nothin'.
$
```

你应该已经看出函数是怎样工作的了。注意到函数的用法和你以前见过的 `exists`、`open`，以及别的“命令”有点类似了吧？其实我只是为了让你容易理解才叫它们“命令”，它们的本质其实就是函数。也就是说，你也可以在自己的脚本中创建你自己的“命令”。

加分习题

为自己写一个函数注意事项以供后续参考。你可以写在一个索引卡片上随时阅读，直到你记住所有的要点为止。注意事项如下：

1. 函数定义是以 `def` 开始的吗？
2. 函数名称是以字符和下划线 `_` 组成的吗？
3. 函数名称是不是紧跟着括号 `(` ？
4. 括号里是否包含参数？多个参数是否以逗号隔开？
5. 参数名称是否有重复？（不能使用重复的参数名）
6. 紧跟着参数的是不是括号和冒号 `):` ？
7. 紧跟着函数定义的代码是否使用了 4 个空格的缩进 (`indent`)？
8. 函数结束的位置是否取消了缩进 (“dedent”)？

当你运行（或者说“使用 use”或者“调用 call”）一个函数时，记得检查下面的要点：

1. 调运函数时是否使用了函数的名称？
2. 函数名称是否紧跟着 `(` ？
3. 括号后有无参数？多个参数是否以逗号隔开？
4. 函数是否以 `)` 结尾？

按照这两份检查表里的内容检查你的练习，直到你不需要检查表为止。

最后，将下面这句话阅读几遍：

“‘运行函数 (run)’、‘调用函数 (call)’、和‘使用函数 (use)’是同一个意思”

习题 19：函数和变量

函数这个概念也许承载了太多的信息量，不过别担心。只要坚持做这些练习，对照上个练习中的检查点检查一遍这次的联系，你最终会明白这些内容的。

有一个你可能没有注意到的细节，我们现在强调一下：函数里边的变量和脚本里边的变量之间是没有连接的。下面的这个练习可以让你对这一点有更多的思考：

```
def cheese_and_crackers(cheese_count, boxes_of_crackers):
    print "You have %d cheeses!" % cheese_count
    print "You have %d boxes of crackers!" % boxes_of_crackers
    print "Man that's enough for a party!"
    print "Get a blanket.\n"


print "We can just give the function numbers directly:"
cheese_and_crackers(20, 30)


print "OR, we can use variables from our script:"
amount_of_cheese = 10
amount_of_crackers = 50

cheese_and_crackers(amount_of_cheese, amount_of_crackers)


print "We can even do math inside too:"
cheese_and_crackers(10 + 20, 5 + 6)


print "And we can combine the two, variables and math:"
cheese_and_crackers(amount_of_cheese + 100, amount_of_crackers + 1000)
```

通过这个练习，你看到我们给我们的函数 `cheese_and_crackers` 很多的参数，然后在函数里把它们打印出来。我们可以在函数里用变量名，我们可以在函数里做运算，我们甚至可以将变量和运算结合起来。

从一方面来说，函数的参数和我们的生成变量时用的 `=` 赋值符类似。事实上，如果一个物件你可以用 `=` 将其命名，你通常也可以将其作为参数传递给一个函数。

你应该看到的结果

你应该研究一下脚本的输出，和你想象的结果对比一下看有什么不同。

```
$ python ex19.py
We can just give the function numbers directly:
You have 20 cheeses!
You have 30 boxes of crackers!
Man that's enough for a party!
Get a blanket.

OR, we can use variables from our script:
You have 10 cheeses!
You have 50 boxes of crackers!
Man that's enough for a party!
Get a blanket.

We can even do math inside too:
You have 30 cheeses!
You have 11 boxes of crackers!
Man that's enough for a party!
Get a blanket.

And we can combine the two, variables and math:
You have 110 cheeses!
You have 1050 boxes of crackers!
Man that's enough for a party!
Get a blanket.
$
```

加分习题

1. 倒着将脚本读完，在每一行上面添加一行注解，说明这行的作用。
2. 从最后一行开始，倒着阅读每一行，读出所有的重要字符来。
3. 自己编至少一个函数出来，然后用 10 种方法运行这个函数。

习题 20：函数和文件

回忆一下函数的要点，然后一边做这节练习，一边注意一下函数和文件是如何在一起协作发挥作用的。

```
from sys import argv

script, input_file = argv

def print_all(f):
    print f.read()

def rewind(f):
    f.seek(0)

def print_a_line(line_count, f):
    print line_count, f.readline()

current_file = open(input_file)

print "First let's print the whole file:\n"

print_all(current_file)

print "Now let's rewind, kind of like a tape."

rewind(current_file)

print "Let's print three lines:"

current_line = 1
print_a_line(current_line, current_file)

current_line = current_line + 1
print_a_line(current_line, current_file)

current_line = current_line + 1
print_a_line(current_line, current_file)
```

特别注意一下，每次运行 print_a_line 时，我们是怎样传递当前的行号信息的。

你应该看到的结果

```
$ python ex20.py test.txt
First let's print the whole file:

To all the people out there.
I say I don't like my hair.
I need to shave it off.

Now let's rewind, kind of like a tape.
Let's print three lines:
1 To all the people out there.

2 I say I don't like my hair.

3 I need to shave it off.

$
```

加分习题

1. 通读脚本，在每行之前加上注解，以理解脚本里发生的事情。
2. 每次 print_a_line 运行时，你都传递了一个叫 current_line 的变量。在每次调用函数时，打印出 current_line 的至，跟踪一下它在 print_a_line 中是怎样变成 line_count 的。
3. 找出脚本中每一个用到函数的地方。检查 def 一行，确认参数没有用错。
4. 上网研究一下 file 中的 seek 函数是做什么用的。试着运行 pydoc file 看看能不能学到更多。
5. 研究一下 += 这个简写操作符的作用，写一个脚本，把这个操作符用在里边试一下。

习题 21：函数可以返回东西

你已经学过使用 = 给变量命名，以及将变量定义为某个数字或者字符串。接下来我们将让你见证更多奇迹。我们要演示给你的是如何使用 = 以及一个新的 Python 词汇 return 来将变量设置为“一个函数的值”。有一点你需要及其注意，不过我们暂且不讲，先撰写下面的脚本吧：

```
def add(a, b):
    print "ADDING %d + %d" % (a, b)
    return a + b

def subtract(a, b):
    print "SUBTRACTING %d - %d" % (a, b)
    return a - b

def multiply(a, b):
    print "MULTIPLYING %d * %d" % (a, b)
    return a * b

def divide(a, b):
    print "DIVIDING %d / %d" % (a, b)
    return a / b


print "Let's do some math with just functions!"

age = add(30, 5)
height = subtract(78, 4)
weight = multiply(90, 2)
iq = divide(100, 2)

print "Age: %d, Height: %d, Weight: %d, IQ: %d" % (age, height, weight, iq)


# A puzzle for the extra credit, type it in anyway.
print "Here is a puzzle."

```

```
what = add(age, subtract(height, multiply(weight, divide(iq, 2))))

print "That becomes: ", what, "Can you do it by hand?"
```

现在我们创建了我们自己的加减乘除数学函数：add, subtract, multiply, 以及 divide。重要的是函数的最后一行，例如 add 的最后一行是 return a + b，它实现的功能是这样的：

1. 我们调用函数时使用了两个参数：a 和 b 。
2. 我们打印出这个函数的功能，这里就是计算加法（adding）
3. 接下来我们告诉 Python 让它做某个回传的动作：我们将 a + b 的值返回 (return)。或者你可以这么说："我将 a 和 b 加起来，再把结果返回。"
4. Python 将两个数字相加，然后当函数结束的时候，它就可以将 a + b 的结果赋予一个变量。

和本书里的很多其他东西一样，你要慢慢消化这些内容，一步一步执行下去，追踪一下究竟发生了什么。为了帮助你理解，本节的加分习题将让你解决一个迷题，并且让你学到点比较酷的东西。

你应该看到的结果

```
$ python ex21.py
Let's do some math with just functions!
ADDING 30 + 5
SUBTRACTING 78 - 4
MULTIPLYING 90 * 2
DIVIDING 100 / 2
Age: 35, Height: 74, Weight: 180, IQ: 50
Here is a puzzle.
DIVIDING 50 / 2
MULTIPLYING 180 * 25
SUBTRACTING 74 - 4500
ADDING 35 + -4426
That becomes:  -4391 Can you do it by hand?
$
```

加分习题

1. 如果你不是很确定 return 的功能，试着自己写几个函数出来，让它们返回一些值。你可以将任何可以放在 = 右边的东西作为一个函数的返回值。
2. 这个脚本的结尾是一个迷题。我将一个函数的返回值用作了另外一个函数的参数。我将它们链接到了一起，就跟写数学等式一样。这样可能有些难读，不过运行一下你就知道结果了。接下来，你需要试试看能不能用正常的方法实现和这个表达式一样的功能。

3. 一旦你解决了这个迷题，试着修改一下函数里的某些部分，然后看会有什么样的结果。你可以有目的地修改它，让它输出另外一个值。
4. 最后，颠倒过来做一次。写一个简单的等式，使用一样的函数来计算它。

这个习题可能会让你有些头大，不过还是慢慢来，把它当做一个游戏，解决这样的迷题正是编程的乐趣之一。后面你还会看到类似的小谜题。

习题 22: 到现在你学到了哪些东西?

这节以及下一节的习题中不会有任何代码，所以也不会有习题答案或者加分习题。其实这节习题可以说是一个巨型的加分习题。我将让你完成一个表格，让你回顾你到现在学到的所有东西。

首先，回到你的每一个习题的脚本里，把你碰到的每一个词和每一个符号（symbol，character 的别名）写下来。确保你的符号列表是完整的。

下一步，在每一个关键词和字符后面写出它的名字，并且说明它的作用。如果你在书里找不到符号的名字，就上网找一下。如果你不知道某个关键字或者符号的作用，就回到用到该字符的章节通读一下，并且在脚本中测试一下这个字符的用处。

你也许会碰到一些横竖找不到答案的东西，只要把这些记在列表里，它可以提示你让你知道还有哪些东西不懂，等下次碰到的时候，你就不会轻易跳过了。

你的列表做好以后，再花几天时间重写一遍这份列表，确认里边的东西都是正确的。你可能觉得这很无聊，不过你还是需要坚持完成任务。

等你记住了这份列表中的所有内容，就试着把这份列表默写一遍。如果发现自己漏掉或者忘记了某些内容，就回去再记一遍。

> **Warning:** 做这节练习没有失败，只有尝试，请牢记这一点。

你学到的东西

这种记忆练习是枯燥无味的，所以知道它的意义很重要。它会让你明确目标，让你知道你所有努力的目的。

在这节练习中你学会的是各种符号的名称，这样读代码对你来说会更加容易。这和学英语时记忆字母表和基本单词的意义是一样的，不同的是 Python 中会有一些你不熟悉的字符。

慢慢做，别让它成为你的负担。这些符号对你来说应该比较熟悉，所以记住它们应该不是很费力的事情。你可以一次花个 15 分钟，然后休息一下。作息结合可以让你学得更快，而且可以让你保持士气。

习题 23：读代码

上一周你应该已经牢记了你的符号列表。现在你需要将这些运用起来，再花一周的时间，在网上阅读代码。这个任务初看会觉得很艰巨。我将直接把你丢到深水区呆几天，让你竭尽全力去读懂实实在在的项目里的代码。这节练习的目的不是让你读懂，而是让你学会下面的技能：

1. 找到你需要的 Python 代码。
2. 通读代码，找到文件。
3. 尝试理解你找到的代码。

以你现在的水平，你还不具备完全理解你找到的代码的能力，不过通过接触这些代码，你可以熟悉真正的编程项目会是什么样子。

当你做这节练习时，你可以把自己当成是一个人类学家来到了一片陌生的大陆，你只懂得一丁点本地语言，但你需要接触当地人并且生存下去。当然做练习不会碰到生存问题，这毕竟这不是荒野或者丛林。

你要做的事情如下：

1. 使用你的浏览器登录 bitbucket.org，搜索 “python”。
2. 忽略那些提到 “Python 3” 的项目，它们只会让你变迷糊。
3. 随便找一个项目，然后点进去。
4. 点击 `Source` 标签，浏览目录和文件列表，直到你看到以 `.py` 结尾的文件（setup.py 就别看了，这样的文件看了也没用）。
5. 从头开始阅读你找到的代码。把它的功能用笔记记下来。
6. 如果你看到一些有趣的符号或者奇怪的字串，你可以把它们记下来，日后再进行研究。

就是这样，你的任务是使用你目前学到的东西，看自己能不能读懂一些代码，看出它们的功能来。你可以先粗略地阅读，然后再细读。也许你还可以试试将难度比较大的部分一字不漏地朗读出来。

现在再试试其它三个站点：

- github.com
- launchpad.net
- koders.com

在这些网站你可能还会看到以 `.c` 结尾的奇怪文件，不过你只需要看 `.py` 结尾的文件就可以了。

最后一个有趣的事情是你可以在这四个网站搜索“python”以外的你感兴趣的话题，例如你可以搜索“journalism（新闻）”，“cooking（厨艺）”，“physics（物理）”，或者任何你感兴趣的话题。你也许会找到一些你对你有用的，可以直接拿来用的代码。

习题 24：更多练习

你离这本书第一部分的结尾已经不远了，你应该已经具备了足够的 Python 基础知识，可以继续学习一些编程的原理了，但你应该做更多的练习。这个练习的内容比较长，它的目的是锻炼你的毅力，下一个习题也差不多是这样的，好好完成它们，做到完全正确，记得仔细检查。

```python
print "Let's practice everything."
print 'You\'d need to know \'bout escapes with \\ that do \n newlines and \t tabs.'

poem = """
\tThe lovely world
with logic so firmly planted
cannot discern \n the needs of love
nor comprehend passion from intuition
and requires an explanation
\n\t\twhere there is none.
"""

print "--------------"
print poem
print "--------------"


five = 10 - 2 + 3 - 6
print "This should be five: %s" % five

def secret_formula(started):
    jelly_beans = started * 500
    jars = jelly_beans / 1000
    crates = jars / 100
    return jelly_beans, jars, crates


start_point = 10000
beans, jars, crates = secret_formula(start_point)

```

```
print "With a starting point of: %d" % start_point
print "We'd have %d beans, %d jars, and %d crates." % (beans, jars, crates)

start_point = start_point / 10

print "We can also do that this way:"
print "We'd have %d beans, %d jars, and %d crates." % secret_formula(start_point)
```

你应该看到的结果

```
$ python ex24.py
Let's practice everything.
You'd need to know 'bout escapes with \ that do
 newlines and          tabs.
--------------

        The lovely world
with logic so firmly planted
cannot discern
 the needs of love
nor comprehend passion from intuition
and requires an explanation

                where there is none.

--------------
This should be five: 5
With a starting point of: 10000
We'd have 5000000 beans, 5000 jars, and 50 crates.
We can also do that this way:
We'd have 500000 beans, 500 jars, and 5 crates.
$
```

加分习题

1. 记得仔细检查结果，从后往前倒着检查，把代码朗读出来，在不清楚的位置加上注释。
2. 故意把代码改错，运行并检查会发生什么样的错误，并且确认你有能力改正这些错误。

习题 25：更多更多的练习

我们将做一些关于函数和变量的练习，以确认你真正掌握了这些知识。这节练习对你来说可以说是一本道：写程序，逐行研究，弄懂它。

不过这节练习还是有些不同，你不需要运行它，取而代之，你需要将它导入到 python 里通过自己执行函数的方式运行。

```
def break_words(stuff):
    """This function will break up words for us."""
    words = stuff.split(' ')
    return words

def sort_words(words):
    """Sorts the words."""
    return sorted(words)

def print_first_word(words):
    """Prints the first word after popping it off."""
    word = words.pop(0)
    print word

def print_last_word(words):
    """Prints the last word after popping it off."""
    word = words.pop(-1)
    print word

def sort_sentence(sentence):
    """Takes in a full sentence and returns the sorted words."""
    words = break_words(sentence)
    return sort_words(words)

def print_first_and_last(sentence):
    """Prints the first and last words of the sentence."""
    words = break_words(sentence)
    print_first_word(words)
    print_last_word(words)
```

```python

def print_first_and_last_sorted(sentence):
    """Sorts the words then prints the first and last one."""
    words = sort_sentence(sentence)
    print_first_word(words)
    print_last_word(words)
```

首先以正常的方式 python ex25.py 运行，找出里边的错误，并把它们都改正过来。然后你需要接着下面的答案章节完成这节练习。

你应该看到的结果

这节练习我们将在你之前用来做算术的 python 编译器里，用交互的方式和你的.py 作交流。

这是我做出来的样子：

```
$ python
Python 2.5.1 (r251:54863, Feb  6 2009, 19:02:12)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import ex25
>>> sentence = "All good things come to those who wait."
>>> words = ex25.break_words(sentence)
>>> words
['All', 'good', 'things', 'come', 'to', 'those', 'who', 'wait.']
>>> sorted_words = ex25.sort_words(words)
>>> sorted_words
['All', 'come', 'good', 'things', 'those', 'to', 'wait.', 'who']
>>> ex25.print_first_word(words)
All
>>> ex25.print_last_word(words)
wait.
>>> wrods
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'wrods' is not defined
>>> words
['good', 'things', 'come', 'to', 'those', 'who']
>>> ex25.print_first_word(sorted_words)
All
>>> ex25.print_last_word(sorted_words)
who
>>> sorted_words
['come', 'good', 'things', 'those', 'to', 'wait.']
>>> sorted_words = ex25.sort_sentence(sentence)
```

```
30 >>> sorted_words
31 ['All', 'come', 'good', 'things', 'those', 'to', 'wait.', 'who']
32 >>> ex25.print_first_and_last(sentence)
33 All
34 wait.
35 >>> ex25.print_first_and_last_sorted(sentence)
36 All
37 who
38 >>> ^D
39 $
```

我们来逐行分析一下每一步实现的是什么：

- 在第 5 行你将自己的 `ex25.py` 执行了 import，和你做过的其它 import 一样。在 import 的时候你不需要加 `.py` 后缀。这个过程里，你把 `ex25.py` 当做了一个“模组 (module)”来使用，你在这个模组里定义的函数也可以直接调用出来。
- 第 6 行你创建了一个用来处理的“句子 (sentence)”。
- 第 7 行你使用 `ex25` 调用你的第一个函数 `ex25.break_words`。其中的 `.` (dot, period) 符号可以告诉 Python：“嗨，我要运行 `ex25` 里的哪个个叫 `break_words` 的函数！”
- 第 8 行我们只是输入 `words`，而 python 将在第 9 行打印出这个变量里边有什么。看上去可能会觉得奇怪，不过这其实是一个“列表 (list)”，你会在后面的章节中学到它。
- 10-11 行我们使用 `ex25.sort_words` 来得到一个排序过的句子。
- 13-16 行我们使用 `ex25.print_first_word` 和 `ex25.print_last_word` 将第一个和最后一个词打印出来。
- 第 17 行比较有趣。我把 `words` 变量写错成了 `wrods`，所以 python 在 18-20 行给了一个错误信息。
- 21-22 行我们打印出了修改过的词汇列表。第一个和最后一个单词我们已经打印过了，所以在这里没有再次打印出来。

剩下的行你需要自己分析一下，就留作你的加分习题了。

加分习题

1. 研究答案中没有分析过的行，找出它们的来龙去脉。确认自己明白了自己使用的是模组 `ex25` 中定义的函数。
2. 试着执行 `help(ex25)` 和 `help(ex25.break_words)` 。这是你得到模组帮助文档的方式。所谓帮助文档就是你定义函数时放在 `"""` 之间的东西，它们也被称作 `documentation comments` （文档注解），后面你还会看到更多类似的东西。
3. 重复键入 `ex25.` 是很烦的一件事情。有一个捷径就是用 `from ex25 import *` 的方式导入模组。这相当于说：“我要把 ex25 中所有的东西 import 进来。”程序员喜欢说这样的倒装句，开一个新的会话，

看看你所有的函数是不是已经在那里了。

4. 把你脚本里的内容逐行通过 `python` 编译器执行，看看会是什么样子。你可以执行 CTRL-D (Windows 下是 CTRL-Z) 来关闭编译器。

习题 26: 恭喜你，现在可以考试了！

你已经差不多完成这本书的前半部分了，不过后半部分才是更有趣的。你将学到逻辑，并通过条件判断实现有用的功能。

在你继续学习之前，你有一道试题要做。这道试题很难，因为它需要你修正别人写的代码。当你成为程序员以后，你将需要经常面对别的程序员的代码，也许还有他们的傲慢态度，他们会经常说自己的代码是完美的。

这样的程序员是自以为是不在乎别人的蠢货。优秀的科学家会对他们自己的工作持怀疑态度，同样，优秀的程序员也会认为自己的代码总有出错的可能，他们会先假设是自己的代码有问题，然后用排除法清查所有可能是自己有问题的地方，最后才会得出"这是别人的错误"这样的结论。

在这节练习中，你将面对一个水平糟糕的程序员，并改好他的代码。我将习题 24 和 25 胡乱拷贝到了一个文件中，随机地删掉了一些字符，然后添加了一些错误进去。大部分的错误是 Python 在执行时会告诉你的，还有一些算术错误是你要自己找出来的。再剩下来的就是格式和拼写错误了。

所有这些错误都是程序员很容易犯的，就算有经验的程序员也不例外。

你的任务是将此文件修改正确，用你所有的技能改进这个脚本。你可以先分析这个文件，或者你还可以把它像学期论文一样打印出来，修正里边的每一个缺陷，重复修正和运行的动作，直到这个脚本可以完美地运行起来。在整个过程中不要寻求帮助，如果你卡在某个地方无法进行下去，那就休息一会晚点再做。

就算你需要几天才能完成，也不要放弃，直到完全改对为止。

最后要说的是，这个练习的目的不是写程序，而是修正现有的程序，你需要访问下面的网站：

- http://learnpythonthehardway.org/exercise26.txt

从那里把代码复制粘贴过来，命名为 `ex26.py`，这也是本书唯一一处允许你复制粘贴的地方。

习题 27: 记住逻辑关系

到此为止你已经学会了读写文件，命令行处理，以及很多 Python 数学运算功能。今天，你将要开始学习逻辑了。你要学习的不是研究院里的高深逻辑理论，只是程序员每天都用到的让程序跑起来的基础逻辑知识。

学习逻辑之前你需要先记住一些东西。这个练习我要求你一个星期完成，不要擅自修改日程，就算你烦得不得了，也要坚持下去。这个练习会让你背下来一系列的逻辑表格，这会让你更容易地完成后面的习题。

需要事先警告你的是：这件事情一开始一点乐趣都没有，你会一开始就觉得它很无聊乏味，但它的目的是教你程序员必须的一个重要技能——一些重要的概念是必须记住的，一旦你明白了这些概念，你会获得相当的成就感，但是一开始你会觉得它们很难掌握，就跟和乌贼摔跤一样，而等到某一天，你会刷的一下豁然开朗。你会从这些基础的记忆学习中得到丰厚的回报。

这里告诉你一个记住某样东西，而不让自己抓狂的方法：在一整天里，每次记忆一小部分，把你最需要加强的部分标记起来。不要想着在两小时内连续不停地背诵，这不会有什么好的结果。不管你花多长时间，你的大脑也只会留住你在前 15 或者 30 分钟内看过的东西。

取而代之，你需要做的是创建一些索引卡片，卡片有两列内容，正面写下逻辑关系，反面写下答案。你需要做到的结果是：拿出一张卡片来，看到正面的表达式，例如 “True or False”，你可以立即说出背面的结果是 “True”！坚持练习，直到你能做到这一点为止。

一旦你能做到这一点了，接下来你需要每天晚上自己在笔记本上写一份真值表出来。不要只是抄写它们，试着默写真值表，如果发现哪里没记住的话，就飞快地撇一眼这里的答案。这样将训练你的大脑让它记住整个真值表。

不要在这上面花超过一周的时间，因为你在后面的应用过程中还会继续学习它们。

逻辑术语

在 python 中我们会用到下面的术语（字符或者词汇）来定义事物的真 (True) 或者假 (False)。计算机的逻辑就是在程序的某个位置检查这些字符或者变量组合在一起表达的结果是真是假。

- `and` 与
- `or` 或
- `not` 非

- != (not equal) 不等于
- == (equal) 等于
- >= (greater-than-equal) 大于等于
- <= (less-than-equal) 小于等于
- True 真
- False 假

其实你已经见过这些字符了，但这些词汇你可能还没见过。这些词汇 (and, or, not) 和你期望的效果其实是一样的，跟英语里的意思一模一样。

真值表

我们将使用这些字符来创建你需要记住的真值表。

NOT	True?
not False	True
not True	False

OR	True?
True or False	True
True or True	True
False or True	True
False or False	False

AND	True?
True and False	False
True and True	True
False and True	False
False and False	False

NOT OR	True?
not (True or False)	False
not (True or True)	False
not (False or True)	False
not (False or False)	True

NOT AND	True?
not (True and False)	True
not (True and True)	False
not (False and True)	True
not (False and False)	True

!=	True?
1 != 0	True
1 != 1	False
0 != 1	True
0 != 0	False

==	True?
1 == 0	False
1 == 1	True
0 == 1	False
0 == 0	True

现在使用这些表格创建你自己的卡片，再花一个星期慢慢记住它们。记住一点，这本书不会要求你成功或者失败，只要每天尽力去学，在尽力的基础上多花一点功夫就可以了。

习题 28: 布尔表达式练习

上一节你学到的逻辑组合的正式名称是“布尔逻辑表达式 (boolean logic expression)”。在编程中，布尔逻辑可以说是无处不在。它们是计算机运算的基础和重要组成部分，掌握它们就跟学音乐掌握音阶一样重要。

在这节练习中，你将在 `python` 里使用到上节学到的逻辑表达式。先为下面的每一个逻辑问题写出你认为的答案，每一题的答案要么为 True 要么为 False。写完以后，你需要将 `python` 运行起来，把这些逻辑语句输入进去，确认你写的答案是否正确。

1. `True and True`
2. `False and True`
3. `1 == 1 and 2 == 1`
4. `"test" == "test"`
5. `1 == 1 or 2 != 1`
6. `True and 1 == 1`
7. `False and 0 != 0`
8. `True or 1 == 1`
9. `"test" == "testing"`
10. `1 != 0 and 2 == 1`
11. `"test" != "testing"`
12. `"test" == 1`
13. `not (True and False)`
14. `not (1 == 1 and 0 != 1)`
15. `not (10 == 1 or 1000 == 1000)`
16. `not (1 != 10 or 3 == 4)`
17. `not ("testing" == "testing" and "Zed" == "Cool Guy")`
18. `1 == 1 and not ("testing" == 1 or 1 == 0)`

19. `"chunky" == "bacon" and not (3 == 4 or 3 == 3)`
20. `3 == 3 and not ("testing" == "testing" or "Python" == "Fun")`

在本节结尾的地方我会给你一个理清复杂逻辑的技巧。

所有的布尔逻辑表达式都可以用下面的简单流程得到结果：

1. 找到相等判断的部分 (== or !=)，将其改写为其最终值 (True 或 False)。
2. 找到括号里的 and/or，先算出它们的值。
3. 找到每一个 not，算出他们反过来的值。
4. 找到剩下的 and/or，解出它们的值。
5. 等你都做完后，剩下的结果应该就是 True 或者 False 了。

下面我们以 #20 逻辑表达式演示一下：

```
3 != 4 and not ("testing" != "test" or "Python" == "Python")
```

接下来你将看到这个复杂表达式是如何逐级解为一个单独结果的：

1. **解出每一个等值判断:**
   (a) `3 != 4` 为 `True`: `True and not ("testing" != "test" or "Python" == "Python")`
   (b) `"testing" != "test"` 为 `True`: `True and not (True or "Python" == "Python")`
   (c) `"Python" == "Python"`: `True and not (True or True)`
2. **找到括号中的每一个 and/or :**
   (a) `(True or True)` 为 True: `True and not (True)`
3. **找到每一个 not 并将其逆转:**
   (a) `not (True)` 为 False: `True and False`
4. **找到剩下的 and/or，解出它们的值:**
   (a) `True and False` 为 False

这样我们就解出了它最终的值为 False.

> **Warning:** 复杂的逻辑表达式一开始看上去可能会让你觉得很难。而且你也许已经碰壁过了，不过别灰心，这些“逻辑体操”式的训练只是让你逐渐习惯起来，这样后面你可以轻易应对编程里边更酷的一些东西。只要你坚持下去，不放过自己做错的地方就行了。如果你暂时不太能理解也没关系，弄懂的时候总会到来的。

你应该看到的结果

以下内容是在你自己猜测结果以后，通过和 `python` 对话得到的结果：

```
$ python
Python 2.5.1 (r251:54863, Feb  6 2009, 19:02:12)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> True and True
True
>>> 1 == 1 and 2 == 2
True
```

加分习题

1. Python 里还有很多和 `!=` 、`==` 类似的操作符. 试着尽可能多地列出 Python 中的等价运算符。例如 `<` 或者 `<=` 就是。
2. 写出每一个等价运算符的名称。例如 `!=` 叫 "not equal（不等于）"。
3. 在 `python` 中测试新的布尔操作。在敲回车前你需要喊出它的结果。不要思考，凭自己的第一感就可以了。把表达式和结果用笔写下来再敲回车，最后看自己做对多少，做错多少。
4. 把习题 3 那张纸丢掉，以后你不再需要查询它了。

习题 29: 如果 (if)

下面是你要写的作业，这段向你介绍了“if 语句”。把这段输入进去，让它能正确执行。然后我们看看你是否有所收获。

```
people = 20
cats = 30
dogs = 15


if people < cats:
    print "Too many cats! The world is doomed!"

if people > cats:
    print "Not many cats! The world is saved!"

if people < dogs:
    print "The world is drooled on!"

if people > dogs:
    print "The world is dry!"


dogs += 5

if people >= dogs:
    print "People are greater than or equal to dogs."

if people <= dogs:
    print "People are less than or equal to dogs."


if people == dogs:
    print "People are dogs."
```

你应该看到的结果

```
$ python ex29.py
Too many cats! The world is doomed!
The world is dry!
People are greater than or equal to dogs.
People are less than or equal to dogs.
People are dogs.
$
```

加分习题

猜猜“if 语句”是什么，它有什么用处。在做下一道习题前，试着用自己的话回答下面的问题:

1. 你认为 `if` 对于它下一行的代码做了什么？
2. 为什么 `if` 语句的下一行需要 4 个空格的缩进？
3. 如果不缩进，会发生什么事情？
4. 把习题 27 中的其它布尔表达式放到 ‘‘if 语句 ‘‘中会不会也可以运行呢？试一下。
5. 如果把变量 `people`, `cats`, 和 `dogs` 的初始值改掉，会发生什么事情？

习题 30: Else 和 If

前一习题中你写了一些“if 语句 (if-statements)”，并且试图猜出它们是什么，以及实现的是什么功能。在你继续学习之前，我给你解释一下上一节的加分习题的答案。上一节的加分习题你做过了吧，有没有？

1. 你认为 `if` 对于它下一行的代码做了什么？If 语句为代码创建了一个所谓的“分支”，就跟 RPG 游戏中的情节分支一样。if 语句告诉你的脚本：“如果这个布尔表达式为真，就运行接下来的代码，否则就跳过这一段。”
2. 为什么 `if` 语句的下一行需要 4 个空格的缩进？行尾的冒号的作用是告诉 Python 接下来你要创建一个新的代码区段。这根你创建函数时的冒号是一个道理。
3. 如果不缩进, 会发生什么事情? 如果你没有缩进，你应该会看到 Python 报错。Python 的规则里，只要一行以“冒号 (colon)”：结尾，它接下来的内容就应该有缩进。
4. 把习题 27 中的其它布尔表达式放到 `if` 语句 中会不会也可以运行呢？试一下。可以。而且不管多复杂都可以，虽然写复杂的东西通常是一种不好的编程风格。
5. 如果把变量 `people`, `cats`, 和 `dogs` 的初始值改掉, 会发生什么事情? 因为你比较的对象是数字，如果你把这些数字改掉的话，某些位置的 if 语句会被演绎为 `True`，而它下面的代码区段将被运行。你可以试着修改这些数字，然后在头脑里假想一下那一段代码会被运行。

把我的答案和你的答案比较一下，确认自己真正懂得代码“区段”的含义。这点对于你下一节的练习很重要，因为你将会写很多的 if 语句。

把这一段写下来，并让它运行起来：

```
people = 30
cars = 40
buses = 15


if cars > people:
    print "We should take the cars."
elif cars < people:
    print "We should not take the cars."
else:
    print "We can't decide."

```

```python
if buses > cars:
    print "That's too many buses."
elif buses < cars:
    print "Maybe we could take the buses."
else:
    print "We still can't decide."

if people > buses:
    print "Alright, let's just take the buses."
else:
    print "Fine, let's stay home then."
```

你应该看到的结果

```
$ python ex30.py
We should take the cars.
Maybe we could take the buses.
Alright, let's just take the buses.
$
```

加分习题

1. 猜想一下 `elif` 和 `else` 的功能。
2. 将 `cars`, `people`, 和 `buses` 的数量改掉，然后追溯每一个 if 语句。看看最后会打印出什么来。
3. 试着写一些复杂的布尔表达式，例如 `cars > people and buses < cars`。
4. 在每一行的上面写注解，说明这一行的功用。

习题 31: 作出决定

这本书的上半部分你打印了一些东西，而且调用了函数，不过一切都是直线式进行的。你的脚本从最上面一行开始，一路运行到结束，但其中并没有决定程序流向的分支点。现在你已经学了 if, else, 和 elif ，你就可以开始创建包含条件判断的脚本了。

上一个脚本中你写了一系列的简单提问测试。这节的脚本中，你将需要向用户提问，依据用户的答案来做出决定。把脚本写下来，多多鼓捣一阵子，看看它的工作原理是什么。

```python
print "You enter a dark room with two doors.  Do you go through door #1 or door #2?"

door = raw_input("> ")

if door == "1":
    print "There's a giant bear here eating a cheese cake.  What do you do?"
    print "1. Take the cake."
    print "2. Scream at the bear."

    bear = raw_input("> ")

    if bear == "1":
        print "The bear eats your face off.  Good job!"
    elif bear == "2":
        print "The bear eats your legs off.  Good job!"
    else:
        print "Well, doing %s is probably better.  Bear runs away." % bear

elif door == "2":
    print "You stare into the endless abyss at Cthulhu's retina."
    print "1. Blueberries."
    print "2. Yellow jacket clothespins."
    print "3. Understanding revolvers yelling melodies."

    insanity = raw_input("> ")

    if insanity == "1" or insanity == "2":
        print "Your body survives powered by a mind of jello.  Good job!"
```

```
    else:
        print "The insanity rots your eyes into a pool of muck.  Good job!"

else:
    print "You stumble around and fall on a knife and die.  Good job!"
```

这里的重点是你可以在“if 语句”内部再放一个“if 语句”。这是一个很强大的功能，可以用来创建嵌套 (nested) 的决定，其中的一个分支将引向另一个分支的子分支。

你需要理解 if 语句包含 if 语句的概念。做一下加分习题，这样你会确信自己真正理解了它们。

你应该看到的结果

我在玩一个小冒险游戏，我玩的水平不怎么好：

```
$ python ex31.py
You enter a dark room with two doors.  Do you go through door #1 or door #2?
> 1
There's a giant bear here eating a cheese cake.  What do you do?
1. Take the cake.
2. Scream at the bear.
> 2
The bear eats your legs off.  Good job!

$ python ex31.py
You enter a dark room with two doors.  Do you go through door #1 or door #2?
> 1
There's a giant bear here eating a cheese cake.  What do you do?
1. Take the cake.
2. Scream at the bear.
> 1
The bear eats your face off.  Good job!

$ python ex31.py
You enter a dark room with two doors.  Do you go through door #1 or door #2?
> 2
You stare into the endless abyss at Cthuhlu's retina.
1. Blueberries.
2. Yellow jacket clothespins.
3. Understanding revolvers yelling melodies.
> 1
Your body survives powered by a mind of jello.  Good job!

$ python ex31.py
You enter a dark room with two doors.  Do you go through door #1 or door #2?
```

```
> 2
You stare into the endless abyss at Cthuhlu's retina.
1. Blueberries.
2. Yellow jacket clothespins.
3. Understanding revolvers yelling melodies.
> 3
The insanity rots your eyes into a pool of muck.  Good job!

$ python ex31.py
You enter a dark room with two doors.  Do you go through door #1 or door #2?
> stuff
You stumble around and fall on a knife and die.  Good job!

$ python ex31.py
You enter a dark room with two doors.  Do you go through door #1 or door #2?
> 1
There's a giant bear here eating a cheese cake.  What do you do?
1. Take the cake.
2. Scream at the bear.
> apples
Well, doing apples is probably better.  Bear runs away.
```

加分习题

为游戏添加新的部分，改变玩家做决定的位置。尽自己的能力扩展这个游戏，不过别把游戏弄得太怪异了。

习题 32：循环和列表

现在你应该有能力写更有趣的程序出来了。如果你能一直跟得上，你应该已经看出将“if 语句”和“布尔表达式”结合起来可以让程序作出一些智能化的事情。

然而，我们的程序还需要能很快地完成重复的事情。这节习题中我们将使用 `for-loop` （for 循环）来创建和打印出各种各样的列表。在做的过程中，你会逐渐明白它们是怎么回事。现在我不会告诉你，你需要自己找到答案。

在你开始使用 for 循环之前，你需要在某个位置存放循环的结果。最好的方法是使用列表 (list)，顾名思义，它就是一个按顺序存放东西的容器。列表并不复杂，你只是要学习一点新的语法。首先我们看看如何创建列表：

```
hairs = ['brown', 'blond', 'red']
eyes = ['brown', 'blue', 'green']
weights = [1, 2, 3, 4]
```

你要做的是以 `[` （左方括号）开头“打开”列表，然后写下你要放入列表的东西，用逗号隔开，就跟函数的参数一样，最后你需要用 `]` （右方括号）结束右方括号的定义。然后 Python 接收这个列表以及里边所有的内容，将其赋给一个变量。

> **Warning:** 对于不会编程的人来说这是一个难点。习惯性思维告诉你的大脑大地是平的。记得上一个练习中的 if 语句嵌套吧，你可能觉得要理解它有些难度，因为生活中一般人不会去像这样的问题，但这样的问题在编程中几乎到处都是。你会看到一个函数调用另外一个包含 if 语句的函数，其中又有嵌套列表的列表。如果你看到这样的东西一时无法弄懂，就用纸币记下来，手动分割下去，直到弄懂为止。

现在我们将使用循环创建一些列表，然后将它们打印出来。

```
the_count = [1, 2, 3, 4, 5]
fruits = ['apples', 'oranges', 'pears', 'apricots']
change = [1, 'pennies', 2, 'dimes', 3, 'quarters']

# this first kind of for-loop goes through a list
for number in the_count:
    print "This is count %d" % number

# same as above
```

```
for fruit in fruits:
    print "A fruit of type: %s" % fruit

# also we can go through mixed lists too
# notice we have to use %r since we don't know what's in it
for i in change:
    print "I got %r" % i

# we can also build lists, first start with an empty one
elements = []

# then use the range function to do 0 to 5 counts
for i in range(0, 6):
    print "Adding %d to the list." % i
    # append is a function that lists understand
    elements.append(i)

# now we can print them out too
for i in elements:
    print "Element was: %d" % i
```

你应该看到的结果

```
$ python ex32.py
This is count 1
This is count 2
This is count 3
This is count 4
This is count 5
A fruit of type: apples
A fruit of type: oranges
A fruit of type: pears
A fruit of type: apricots
I got 1
I got 'pennies'
I got 2
I got 'dimes'
I got 3
I got 'quarters'
Adding 0 to the list.
Adding 1 to the list.
Adding 2 to the list.
Adding 3 to the list.
```

```
Adding 4 to the list.
Adding 5 to the list.
Element was: 0
Element was: 1
Element was: 2
Element was: 3
Element was: 4
Element was: 5
$
```

加分习题

1. 注意一下 `range` 的用法。查一下 `range` 函数并理解它。
2. 在第 22 行，你可以可以直接将 elements 赋值为 range(0,6)，而无需使用 for 循环？
3. 在 Python 文档中找到关于列表的内容，仔细阅读以下，除了 `append` 以外列表还支持哪些操作？

习题 33: While 循环

接下来是一个更在你意料之外的概念：while-loop``（while 循环）。``while-loop 会一直执行它下面的代码片段，直到它对应的布尔表达式为 False 时才会停下来。

等等，你还能跟得上这些术语吧？如果你的某一行是以 :（冒号, colon）结尾，那就意味着接下来的内容是一个新的代码片段，新的代码片段是需要被缩进的。只有将代码用这样的方式格式化，Python 才能知道你的目的。如果你不太明白这一点，就回去看看“if 语句”和“函数”的章节，直到你明白为止。

接下来的练习将训练你的大脑去阅读这些结构化的代码。这和我们将布尔表达式烧录到你的大脑中的过程有点类似。

回到 while 循环，它所作的和 if 语句类似，也是去检查一个布尔表达式的真假，不一样的是它下面的代码片段不是只被执行一次，而是执行完后再调回到 while 所在的位置，如此重复进行，直到 while 表达式为 False 为止。

While 循环有一个问题，那就是有时它会永不结束。如果你的目的是循环到宇宙毁灭为止，那这样也挺好的，不过其他的情况下你的循环总需要有一个结束点。

为了避免这样的问题，你需要遵循下面的规定：

1. 尽量少用 while-loop，大部分时候 for-loop 是更好的选择。
2. 重复检查你的 while 语句，确定你测试的布尔表达式最终会变成 False 。
3. 如果不确定，就在 while-loop 的结尾打印出你要测试的值。看看它的变化。

在这节练习中，你将通过上面的三样事情学会 while-loop :

```
i = 0
numbers = []

while i < 6:
    print "At the top i is %d" % i
    numbers.append(i)

    i = i + 1
    print "Numbers now: ", numbers
    print "At the bottom i is %d" % i

```

```
12
13 print "The numbers: "
14
15 for num in numbers:
16     print num
```

你应该看到的结果

```
$ python ex33.py
At the top i is 0
Numbers now:  [0]
At the bottom i is 1
At the top i is 1
Numbers now:  [0, 1]
At the bottom i is 2
At the top i is 2
Numbers now:  [0, 1, 2]
At the bottom i is 3
At the top i is 3
Numbers now:  [0, 1, 2, 3]
At the bottom i is 4
At the top i is 4
Numbers now:  [0, 1, 2, 3, 4]
At the bottom i is 5
At the top i is 5
Numbers now:  [0, 1, 2, 3, 4, 5]
At the bottom i is 6
The numbers:
0
1
2
3
4
5
```

加分习题

1. 将这个 while 循环改成一个函数，将测试条件 (`i < 6`) 中的 6 换成一个变量。
2. 使用这个函数重写你的脚本，并用不同的数字进行测试。
3. 为函数添加另外一个参数，这个参数用来定义第 8 行的加值 `+ 1` ，这样你就可以让它任意加值了。
4. 再使用该函数重写一遍这个脚本。看看效果如何。

5. 接下来使用 `for-loop` 和 `range` 把这个脚本再写一遍。你还需要中间的加值操作吗？如果你不去掉它，会有什么样的结果？

很有可能你会碰到程序跑着停不下来了，这时你只要按着 `CTRL` 再敲 `c` (CTRL-c)，这样程序就会中断下来了。

习题 34：访问列表的元素

列表的用处很大，但只有你能访问里边的内容时它才能发挥出作用来。你已经学会了按顺序读出列表的内容，但如果你要得到第 5 个元素该怎么办呢？你需要知道如何访问列表中的元素。访问第一个元素的方法是这样的：

```
animals = ['bear', 'tiger', 'penguin', 'zebra']
bear = animals[0]
```

你定义一个 animals 的列表，然后你用 0 来获取第一个元素?! 这是怎么回事啊？因为数学里边就是这样，所以 Python 的列表也是从 0 开始的。虽然看上去很奇怪，这样定义其实有它的好处，而且实际上设计成 0 或者 1 开头其实都可以，

最好的解释方式是将你平时使用数字的方式和程序员使用数字的方式做对比。

假设你在观看上面列表中的四种动物 (`['bear', 'tiger', 'penguin', 'zebra']`) 的赛跑，而它们比赛的名词正好跟列表里的次序一样。这是一场很激动人心的比赛，因为这些动物没打算吃掉对方，而且比赛还真的举办起来了。结果你的朋友来晚了，他想知道谁赢了比赛，他会问你“嘿，谁是第 0 名”吗？不会的，他会问“嘿，谁是第 1 名？”

这是因为动物的次序是很重要的。没有第一个就没有第二个，没有第二个也没有第三个。第零个是不存在的，因为零的意思是什么都没有。“什么都没有”怎么赢比赛嘛，完全不合逻辑。这样的数字我们称之为“序数 (ordinal number)”，因为它们表示的是事物的顺序。

而程序员不能用这种方式思考问题，因为他们可以从列表的任何一个位置取出一个元素来。对程序员来说，上述的列表更像是一叠卡片。如果他们想要 tiger，就抓它出来，如果想要 zebra，也一样抓取出来。要随机地抓取列表里的内容，列表的每一个元素都应该有一个地址，或者一个“index（索引）”，而最好的方式是使用以 0 开头的索引。相信我说的这一点吧，这种方式获取元素会更容易。这类的数字被称为“基数 (cardinal number)”，它意味着你可以任意抓取元素，所以我们需要一个 0 号元素。

那么，这些知识对于你的列表操作有什么帮助呢？很简单，每次你对自己说“我要第 3 只动物”时，你需要将“序数”转换成“基数”，只要将前者减 1 就可以了。第 3 只动物的索引是 2，也就是 penguin。由于你一辈子都在跟序数打交道，所以你需要用这种方式来获得基数，只要减 1 就都搞定了。

记住: ordinal == 有序，以 1 开始；cardinal == 随机选取, 以 0 开始。

让我们练习一下。定义一个动物列表，然后跟着做后面的练习，你需要写出所指位置的动物名称。如果我用的是“1st, 2nd”等说法，那说明我用的是序数，所以你需要减去 1。如果我给你的是基数（0, 1, 2），你只要直接使用即可。.. code-block:: python

animals = ['bear', 'python', 'peacock', 'kangaroo', 'whale', 'platypus']

1. The animal at 1.
2. The 3rd animal.
3. The 1st animal.
4. The animal at 3.
5. The 5th animal.
6. The animal at 2.
7. The 6th animal.
8. The animal at 4.

对于上述每一条，以这样的格式写出一个完整的句子：“The 1st animal is at 0 and is a bear.”然后倒过来念：“The animal at 0 is the 1st animal and is a bear.”

使用 python 检查你的答案。

加分习题

1. 上网搜索一下关于序数 (ordinal number) 和基数 (cardinal number) 的知识并阅读一下。
2. 以你对于这些不同的数字类型的了解，解释一下为什么 “January 1, 2010” 里是 2010 而不是 2009？（提示：你不能随机挑选年份。）
3. 再写一些列表，用一样的方式作出索引，确认自己可以在两种数字之间互相翻译。
4. 使用 python 检查自己的答案。

> **Warning:** 会有程序员告诉你让你去阅读一个叫“Dijkstra”的人写的关于数字的话题。我建议你还是不读为妙。除非你喜欢听一个在编程这一行刚兴起时就停止从事编程了的人对你大喊大叫。

习题 35：分支和函数

你已经学会了 if 语句、函数、还有列表。现在你要练习扭转一下思维了。把下面的代码写下来，看你是否能弄懂它实现的是什么功能。

```
from sys import exit

def gold_room():
    print "This room is full of gold.  How much do you take?"

    next = raw_input("> ")
    if "0" in next or "1" in next:
        how_much = int(next)
    else:
        dead("Man, learn to type a number.")

    if how_much < 50:
        print "Nice, you're not greedy, you win!"
        exit(0)
    else:
        dead("You greedy bastard!")


def bear_room():
    print "There is a bear here."
    print "The bear has a bunch of honey."
    print "The fat bear is in front of another door."
    print "How are you going to move the bear?"
    bear_moved = False

    while True:
        next = raw_input("> ")

        if next == "take honey":
            dead("The bear looks at you then slaps your face off.")
        elif next == "taunt bear" and not bear_moved:
            print "The bear has moved from the door. You can go through it now."
```

```
            bear_moved = True
        elif next == "taunt bear" and bear_moved:
            dead("The bear gets pissed off and chews your leg off.")
        elif next == "open door" and bear_moved:
            gold_room()
        else:
            print "I got no idea what that means."


def cthulhu_room():
    print "Here you see the great evil Cthulhu."
    print "He, it, whatever stares at you and you go insane."
    print "Do you flee for your life or eat your head?"

    next = raw_input("> ")

    if "flee" in next:
        start()
    elif "head" in next:
        dead("Well that was tasty!")
    else:
        cthulhu_room()


def dead(why):
    print why, "Good job!"
    exit(0)

def start():
    print "You are in a dark room."
    print "There is a door to your right and left."
    print "Which one do you take?"

    next = raw_input("> ")

    if next == "left":
        bear_room()
    elif next == "right":
        cthulhu_room()
    else:
        dead("You stumble around the room until you starve.")


start()
```

你应该看到的结果

下面是我玩游戏的过程：

```
$ python ex35.py
You are in a dark room.
There is a door to your right and left.
Which one do you take?
> left
There is a bear here.
The bear has a bunch of honey.
The fat bear is in front of another door.
How are you going to move the bear?
> taunt bear
The bear has moved from the door. You can go through it now.
> open door
This room is full of gold.  How much do you take?
> asf
Man, learn to type a number. Good job!
$
```

加分习题

1. 把这个游戏的地图画出来，把自己的路线也画出来。
2. 改正你所有的错误，包括拼写错误。
3. 为你不懂的函数写注解。记得文档注解该怎么写吗？
4. 为游戏添加更多元素。通过怎样的方式可以简化并且扩展游戏的功能呢？
5. 这个 `gold_room` 游戏使用了奇怪的方式让你键入一个数字。这种方式会导致什么样的 bug？你可以用比检查 0、1 更好的方式判断输入是否是数字吗？`int()` 这个函数可以给你一些头绪。

习题 36：设计和调试

现在你已经学会了“if 语句”，我将给你一些使用“for 循环”和“while 循环”的规则，一面你日后碰到麻烦。我还会教你一些调试的小技巧，以便你能发现自己程序的问题。最后，你将需要设计一个和上节类似的小游戏，不过内容略有更改。

If 语句的规则

1. 每一个“if 语句”必须包含一个 `else`.
2. 如果这个 `else` 永远都不应该被执行到，因为它本身没有任何意义，那你必须在 else 语句后面使用一个叫做 die 的函数，让它打印出错误信息并且死给你看，这和上一节的习题类似，这样你可以找到很多的错误。
3. “if 语句”的嵌套不要超过 2 层，最好尽量保持只有 1 层。这意味着如果你在 if 里边又有了一个 if，那你就需要把第二个 if 移到另一个函数里面。
4. 将“if 语句”当做段落来对待，其中的每一个 `if`，`elif`，`else` 组合就跟一个段落的句子组合一样。在这种组合的最前面和最后面留一个空行以作区分。
5. 你的布尔测试应该很简单，如果它们很复杂的话，你需要将它们的运算事先放到一个变量里，并且为变量取一个好名字。

如果你遵循上面的规则，你就会写出比大部分程序员都好的代码来。回到上一个练习中，看看我有没有遵循这些规则，如果没有的话，就将其改正过来。

> **Warning:** 在日常编程中不要成为这些规则的奴隶。在训练中，你需要通过这些规则的应用来巩固你学到的知识，而在实际编程中这些规则有时其实很蠢。如果你觉得哪个规则很蠢，就别使用它。

循环的规则

1. 只有在循环永不停止时使用“while 循环”，这意味着你可能永远都用不到。这条只有 Python 中成立，其他的语言另当别论。
2. 其他类型的循环都使用“for 循环”，尤其是在循环的对象数量固定或者有限的情况下。

调试 (debug) 的小技巧

1. 不要使用 “debugger”。Debugger 所作的相当于对病人的全身扫描。你并不会得到某方面的有用信息，而且你会发现它输出的信息态度，而且大部分没有用，或者只会让你更困惑。
2. 最好的调试程序的方法是使用 `print` 在各个你想要检查的关键环节将关键变量打印出来，从而检查哪里是否有错。
3. 让程序一部分一部分地运行起来。不要等一个很长的脚本写完后才去运行它。写一点，运行一点，再修改一点。

家庭作业

写一个和上节练习类似的游戏。同类的任何题材的游戏都可以，花一个星期让它尽可能有趣一些。作为加分习题，你可以尽量多使用列表、函数、以及模组（记得习题 13 吗？），而且尽量多弄一些新的 Python 代码让你的游戏跑起来。

不过有一点需要注意，你应该把游戏的设计先写出来。在你写代码之前，你应该设计出游戏的地图，创建出玩家会碰到的房间、怪物、以及陷阱等环节。

一旦搞定了地图，你就可以写代码了。如果你发现地图有问题，就调整一下地图，让代码和地图互相符合。

最后一个建议：每一个程序员在开始一个新的大项目时，都会被非理性的恐惧影响到。为了避免这种恐惧，他们会拖延时间，到最后一事无成。我有时会这样，每个人都会有这样的经历，避免这种情况的最好的方法是把自己要做的事情列出来，一次完成一样。

开始做吧。先做一个小一点的版本，扩充它让它变大，把自己要完成的事情一一列出来，然后逐个完成就可以了。

习题 37：复习各种符号

现在该复习你学过的符号和 python 关键字了，而且你在本节还会学到一些新的东西。我在这里所作的是将所有的 Python 符号和关键字列出来，这些都是值得掌握的重点。

在这节课中，你需要复习每一个关键字，从记忆中想起它的作用并且写下来，接着上网搜索它真正的功能。有些内容可能是无法搜索的，所以这对你可能有些难度，不过你还是需要坚持尝试。

如果你发现记忆中的内容有误，就在索引卡片上写下正确的定义，试着将自己的记忆纠正过来。如果你就是不知道它的定义，就把它也直接写下来，以后再做研究。

最后，将每一种符号和关键字用在程序里，你可以用一个小程序来做，也可以尽量多谢一些程序来巩固记忆。这里的关键点是明白各个符号的作用，确认自己没搞错，如果搞错了就纠正过来，然后将其用在程序里，并且通过这样的方式巩固自己的记忆。

Keywords（关键字）

- `and`
- `del`
- `from`
- `not`
- `while`
- `as`
- `elif`
- `global`
- `or`
- `with`
- `assert`
- `else`
- `if`

- pass
- yield
- break
- except
- import
- print
- class
- exec
- in
- raise
- continue
- finally
- is
- return
- def
- for
- lambda
- try

数据类型

针对每一种数据类型，都举出一些例子来，例如针对 string，你可以举出一些字符串，针对 number，你可以举出一些数字。

- True
- False
- None
- strings
- numbers
- floats
- lists

字符串转义序列 (Escape Sequences)

对于字符串转义序列，你需要再字符串中应用它们，确认自己清楚地知道它们的功能。

- \\
- \'
- \"
- \a
- \b
- \f
- \n
- \r
- \t
- \v

字符串格式化 (String Formats)

一样的，在字符串中使用它们，确认它们的功能。

- %d
- %i
- %o
- %u
- %x
- %X
- %e
- %E
- %f
- %F
- %g
- %G
- %c
- %r

- %s
- %%

操作符号

有些操作符号你可能还不熟悉，不过还是一一看过去，研究一下它们的功能，如果你研究不出来也没关系，记录下来日后解决。

- +
- -
- *
- **
- /
- //
- %
- <
- >
- <=
- >=
- ==
- !=
- <>
- ( )
- [ ]
- { }
- @
- ,
- :
- .
- =
- ;
- +=

- `-=`
- `*=`
- `/=`
- `//=`
- `%=`
- `**=`

花一个星期学习这些东西，如果你能提前完成就更好了。我们的目的是覆盖到所有的符号类型，确认你已经牢牢记住它们。另外很重要的一点是这样你可以找出自己还不知道哪些东西，为自己日后学习找到一些方向。

习题 38：阅读代码

现在去找一些 Python 代码阅读一下。你需要自己找代码，然后从中学习一些东西。你学到的东西已经足够让你看懂一些代码了，但你可能还无法理解这些代码的功能。这节课我要教给你的是：如何运用你学到的东西理解别人的代码。

首先把你想要理解的代码打印到纸上。没错，你需要打印出来，因为和屏幕输出相比，你的眼睛和大脑更习惯于接受纸质打印的内容。一次最多打印几页就可以了。

然后通读你打印出来的代码并做好标记，标记的内容包括以下几个方面：

1. 函数以及函数的功能。
2. 每个变量的初始赋值。
3. 每个在程序的各个部分中多次出现的变量。它们以后可能会给你带来麻烦。
4. 任何不包含 else 的 `if` 语句。它们是正确的吗？
5. 任何可能没有结束点的 `while` 循环。
6. 最后一条，代码中任何你看不懂的部分都记下来。

接下来你需要通过注解的方式向自己解释代码的含义。解释各个函数的使用方法，各个变量的用途，以及任何其它方面的内容，只要能帮助你理解代码即可。

最后，在代码中比较难的各个部分，逐行或者逐个函数跟踪变量值。你可以再打印一份出来，在空白处写出你要“追踪”的每个变量的值。

一旦你基本理解了代码的功能，回到电脑面前，在屏幕上重读一次，看看能不能找到新的问题点。然后继续找新的代码，用上述的方法去阅读理解，直到你不再需要纸质打印为止。

加分习题

1. 研究一下什么是“流程图 (flow chart)”，并学着画一下。
2. 如果你在读代码的时候找出了错误，试着把它们改对，并把修改内容发给作者。
3. 不使用纸质打印时，你可以使用注解符号 `#` 在程序中加入笔记。有时这些笔记会对后来的读代码的人有很大的帮助。

习题 39：列表的操作

你已经学过了列表。在你学习 “while 循环” 的时候，你对列表进行过 “追加 (append)” 操作，而且将列表的内容打印了出来。另外你应该还在加分习题里研究过 Python 文档，看了列表支持的其他操作。这已经是一段时间以前了，所以如果你不记得了的话，就回到本书的前面再复习一遍把。

找到了吗？还记得吗？很好。那时候你对一个列表执行了 `append` 函数。不过，你也许还没有真正明白发生的事情，所以我们再来看看我们可以对列表进行什么样的操作。

当你看到像 `mystuff.append('hello')` 这样的代码时，你事实上已经在 Python 内部激发了一个连锁反应。以下是它的工作原理：

1. Python 看到你用到了 `mystuff` ，于是就去找到这个变量。也许它需要倒着检查看你有没有在哪里用 `=` 创建过这个变量，或者检查它是不是一个函数参数，或者看它是不是一个全局变量。不管哪种方式，它得先找到 `mystuff` 这个变量才行。

2. 一旦它找到了 `mystuff` ，就轮到处理句点 `.` (period) 这个操作符，而且开始查看 `mystuff` 内部的一些变量了。由于 `mystuff` 是一个列表，Python 知道 `mystuff` 支持一些函数。

3. 接下来轮到了处理 `append` 。Python 会将 “append” 和 `mystuff` 支持的所有函数的名称一一对比，如果确实其中有一个叫 append 的函数，那么 Python 就会去使用这个函数。

4. 接下来 Python 看到了括号 `(` (parenthesis) 并且意识到，“噢，原来这应该是一个函数”，到了这里，它就正常会调用这个函数了，不过这里的函数还要多一个参数才行。

5. 这个额外的参数其实是……`mystuff`! 我知道，很奇怪是不是？不过这就是 Python 的工作原理，所以还是记住这一点，就当它是正常的好了。真正发生的事情其实是 `append(mystuff, 'hello')` ，不过你看到的只是 `mystuff.append('hello')` 。

大部分时候你不需要知道这些细节，不过如果你看到一个像这样的 Python 错误信息的时候，上面的细节就对你有用了：

```
$ python
Python 2.6.5 (r265:79063, Apr 16 2010, 13:57:41)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> class Thing(object):
...     def test(hi):
...             print "hi"
```

```
...
>>> a = Thing()
>>> a.test("hello")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: test() takes exactly 1 argument (2 given)
>>>
```

就是这个吗？嗯，这个是我在 Python 命令行下展示给你的一点魔法。你还没有见过 `class` 不过后面很快就要碰到了。现在你看到 Python 说 `test() takes exactly 1 argument (2 given)` (test() 只可以接受 1 个参数，实际上给了两个)。它意味着 python 把 `a.test("hello")` 改成了 `test(a, "hello")` ，而有人弄错了，没有为它添加 `a` 这个参数。

一下子要消化这么多可能有点难度，不过我们将做几个练习，让你头脑中有一个深刻的印象。下面的练习将字符串和列表混在一起，看看你能不能在里边找出点乐子来：

```python
ten_things = "Apples Oranges Crows Telephone Light Sugar"

print "Wait there's not 10 things in that list, let's fix that."

stuff = ten_things.split(' ')
more_stuff = ["Day", "Night", "Song", "Frisbee", "Corn", "Banana", "Girl", "Boy"]

while len(stuff) != 10:
    next_one = more_stuff.pop()
    print "Adding: ", next_one
    stuff.append(next_one)
    print "There's %d items now." % len(stuff)

print "There we go: ", stuff

print "Let's do some things with stuff."

print stuff[1]
print stuff[-1] # whoa! fancy
print stuff.pop()
print ' '.join(stuff) # what? cool!
print '#'.join(stuff[3:5]) # super stellar!
```

你应该看到的结果

```
$ python ex39.py

Wait there's not 10 things in that list, let's fix that.
```

```
Adding:  Boy
There's 7 items now.
Adding:  Girl
There's 8 items now.
Adding:  Banana
There's 9 items now.
Adding:  Corn
There's 10 items now.
There we go:  ['Apples', 'Oranges', 'Crows', 'Telephone', 'Light', 'Sugar',
      'Boy', 'Girl', 'Banana', 'Corn']
Let's do some things with stuff.
Oranges
Corn
Corn
Apples Oranges Crows Telephone Light Sugar Boy Girl Banana
Telephone#Light
```

加分习题

1. 将每一个被调用的函数以上述的方式翻译成 Python 实际执行的动作。例如：`' '.join(things)` 其实是 `join(' ', things)` 。
2. 将这两种方式翻译为自然语言。例如，`' '.join(things)` 可以翻译成“用‘‘ 连接 (join) things”，而 `join(' ', things)` 的意思是“为‘‘ 和 things 调用 join 函数”。这其实是同一件事情。
3. 上网阅读一些关于“面向对象编程 (Object Oriented Programming)”的资料。晕了吧？嗯，我以前也是。别担心。你将从这本书学到足够用的关于面向对象编程的基础知识，而以后你还可以慢慢学到更多。
4. 查一下 Python 中的 “class” 是什么东西。不要阅读关于其他语言的 “class” 的用法，这会让你更糊涂。
5. `dir(something)` 和 `something` 的 class 有什么关系？
6. 如果你不知道我讲的是些什么东西，别担心。程序员为了显得自己聪明，于是就发明了 Opject Oriented Programming，简称为 OOP，然后他们就开始滥用这个东西了。如果你觉得这东西太难，你可以开始学一下“函数编程 (functional programming)”。

习题 40：字典，可爱的字典

接下来我要教你另外一种让你伤脑筋的容器型数据结构，因为一旦你学会这种容器，你将拥有超酷的能力。这是最有用的容器：字典 (dictionary)。

Python 将这种数据类型叫做 “dict”，有的语言里它的名称是 “hash”。这两种名字我都会用到，不过这并不重要，重要的是它们和列表的区别。你看，针对列表你可以做这样的事情：

```
>>> things = ['a', 'b', 'c', 'd']
>>> print things[1]
b
>>> things[1] = 'z'
>>> print things[1]
z
>>> print things
['a', 'z', 'c', 'd']
>>>
```

你可以使用数字作为列表的索引，也就是你可以通过数字找到列表中的元素。而 `dict` 所作的，是让你可以通过任何东西找到元素，不只是数字。是的，字典可以将一个物件和另外一个东西关联，不管它们的类型是什么，我们来看看：

```
>>> stuff = {'name': 'Zed', 'age': 36, 'height': 6*12+2}
>>> print stuff['name']
Zed
>>> print stuff['age']
36
>>> print stuff['height']
74
>>> stuff['city'] = "San Francisco"
>>> print stuff['city']
San Francisco
>>>
```

你将看到除了通过数字以外，我们还可以用字符串来从字典中获取 `stuff` ，我们还可以用字符串来往字典中添加元素。当然它支持的不只有字符串，我们还可以做这样的事情：

```
>>> stuff[1] = "Wow"
>>> stuff[2] = "Neato"
>>> print stuff[1]
Wow
>>> print stuff[2]
Neato
>>> print stuff
{'city': 'San Francisco', 2: 'Neato',
    'name': 'Zed', 1: 'Wow', 'age': 36,
    'height': 74}
>>>
```

在这里我使用了两个数字。其实我可以使用任何东西，不过这么说并不准确，不过你先这么理解就行了。

当然了，一个只能放东西进去的字典是没啥意思的，所以我们还要有删除物件的方法，也就是使用 del 这个关键字：

```
>>> del stuff['city']
>>> del stuff[1]
>>> del stuff[2]
>>> stuff
{'name': 'Zed', 'age': 36, 'height': 74}
>>>
```

接下来我们要做一个练习，你必须非常仔细，我要求你将这个练习写下来，然后试着弄懂它做了些什么。这个练习很有趣，做完以后你可能会有豁然开朗的感觉。

```
cities = {'CA': 'San Francisco', 'MI': 'Detroit',
                     'FL': 'Jacksonville'}

cities['NY'] = 'New York'
cities['OR'] = 'Portland'

def find_city(themap, state):
    if state in themap:
        return themap[state]
    else:
        return "Not found."

# ok pay attention!
cities['_find'] = find_city

while True:
    print "State? (ENTER to quit)",
    state = raw_input("> ")

```

```
    if not state: break

    # this line is the most important ever! study!
    city_found = cities['_find'](cities, state)
    print city_found
```

Warning: 注意到我用了 `themap` 而不是 `map` 了吧？这是因为 Python 已经有一个函数称作 map 了，所以如果你用 map 做变量名，你后面可能会碰到问题。

你应该看到的结果

```
$ python ex40.py
State? (ENTER to quit) > CA
San Francisco
State? (ENTER to quit) > FL
Jacksonville
State? (ENTER to quit) > O
Not found.
State? (ENTER to quit) > OR
Portland
State? (ENTER to quit) > VT
Not found.
State? (ENTER to quit) >
```

加分习题

1. 在 Python 文档中找到 dictionary (又被称作 dicts, dict) 的相关的内容，学着对 dict 做更多的操作。
2. 找出一些 dict 无法做到的事情。例如比较重要的一个就是 dict 的内容是无序的，你可以检查一下看看是否真是这样。
3. 试着把 `for-loop` 执行到 dict 上面，然后试着在 for-loop 中使用 dict 的 `items()` 函数，看看会有什么样的结果。

习题 41：来自 Percal 25 号行星的哥顿人 (Gothons)

你在上一节中发现 dict 的秘密功能了吗？你可以解释给自己吗？让我来给你解释一下，顺便和你自己的理解对比看有什么不同。这里是我们要讨论的代码：

```
cities['_find'] = find_city
city_found = cities['_find'](cities, state)
```

你要记住一个函数也可以作为一个变量，“def find_city“ 比如这一句创建了一个你可以在任何地方都能使用的变量。在这段代码里，我们首先把函数 `find_city` 放到叫做 `cities` 的字典中，并将其标记为 `'_find'`。这和我们将州和市关联起来的代码做的事情一样，只不过我们在这里放了一个函数的名称。

好了，所以一旦我们知道 `find_city` 是在字典中 `_find` 的位置，这就意味着我们可以去调用它。第二行代码可以分解成如下步骤：

1. Python 看到 `city_found =` 于是知道了需要创建一个变量。
2. 然后它读到 `cities` ，然后知道了它是一个字典
3. 然后看到了 `['_find']` ，于是 Python 就从索引找到了字典 `cities` 中对应的位置，并且获取了该位置的内容。
4. `['_find']` 这个位置的内容是我们的函数 `find_city` ，所以 Python 就知道了这里表示一个函数，于是当它碰到 `(` 就开始了函数调用。
5. `cities, state` 这两个参数将被传递到函数 `find_city` 中，然后这个函数就被运行了。
6. `find_city` 接着从 `cities` 中寻找 `states` ，并且返回它找到的内容，如果什么都没找到，就返回一个信息说它什么都没找到。
7. Python `find_city` 接受返回的信息，最后将该信息赋值给一开始的 `city_found` 这个变量。

我再教你一个小技巧。如果你倒着阅读的话，代码可能会变得更容易理解。让我们来试一下，一样是那行：

1. `state` 和 `city` 是...
2. 作为参数传递给...

3. 一个函数，位置在…
4. '_find' 然后寻找，目的地为…
5. cities 这个位置…
6. 最后赋值给 city_found.

还有一种方法读它，这回是“由里向外”。

1. 找到表达式的中心位置，此次为 ['_find'].
2. 逆时针追溯，首先看到的是一个叫 cities 的字典，这样就知道了 cities 中的 _find 元素。
3. 上一步得到一个函数。继续逆时针寻找，看到的是参数。
4. 参数传递给函数后，函数会返回一个值。然后再逆时针寻找。
5. 最后，我们到了 city_found = 的赋值位置，并且得到了最终结果。

数十年的编程下来，我在读代码的过程中已经用不到上面的三种方法了。我只要瞟一眼就能知道它的意思。甚至给我一整页的代码，我也可以一眼瞄出里边的 bug 和错误。这样的技能是花了超乎常人的时间和精力才锻炼得来的。在磨练的过程中，我学会了下面三种读代码的方法，它们是用户几乎所有的编程语言：

1. 从前向后。
2. 从后向前。
3. 逆时针方向。

下次碰到难懂的语句时，你可以试试这三种方法。

现在我们来写这次的练习，写完后再过一遍，这节习题其实挺有趣的。

```python
from sys import exit
from random import randint

def death():
    quips = ["You died.  You kinda suck at this.",
             "Nice job, you died ...jackass.",
             "Such a luser.",
             "I have a small puppy that's better at this."]

    print quips[randint(0, len(quips)-1)]
    exit(1)


def central_corridor():
    print "The Gothons of Planet Percal #25 have invaded your ship and destroyed"
    print "your entire crew.  You are the last surviving member and your last"
    print "mission is to get the neutron destruct bomb from the Weapons Armory,"
    print "put it in the bridge, and blow the ship up after getting into an "
    print "escape pod."
```

```
    print "\n"
    print "You're running down the central corridor to the Weapons Armory when"
    print "a Gothon jumps out, red scaly skin, dark grimy teeth, and evil clown costume"
    print "flowing around his hate filled body.  He's blocking the door to the"
    print "Armory and about to pull a weapon to blast you."

    action = raw_input("> ")

    if action == "shoot!":
        print "Quick on the draw you yank out your blaster and fire it at the Gothon."
        print "His clown costume is flowing and moving around his body, which throws"
        print "off your aim.  Your laser hits his costume but misses him entirely.  This"
        print "completely ruins his brand new costume his mother bought him, which"
        print "makes him fly into an insane rage and blast you repeatedly in the face until"
        print "you are dead.  Then he eats you."
        return 'death'

    elif action == "dodge!":
        print "Like a world class boxer you dodge, weave, slip and slide right"
        print "as the Gothon's blaster cranks a laser past your head."
        print "In the middle of your artful dodge your foot slips and you"
        print "bang your head on the metal wall and pass out."
        print "You wake up shortly after only to die as the Gothon stomps on"
        print "your head and eats you."
        return 'death'

    elif action == "tell a joke":
        print "Lucky for you they made you learn Gothon insults in the academy."
        print "You tell the one Gothon joke you know:"
        print "Lbhe zbgure vf fb sng, jura fur fvgf nebhaq gur ubhfr, fur fvgf nebhaq gur ubhfr."
        print "The Gothon stops, tries not to laugh, then busts out laughing and can't move."
        print "While he's laughing you run up and shoot him square in the head"
        print "putting him down, then jump through the Weapon Armory door."
        return 'laser_weapon_armory'

    else:
        print "DOES NOT COMPUTE!"
        return 'central_corridor'

def laser_weapon_armory():
    print "You do a dive roll into the Weapon Armory, crouch and scan the room"
    print "for more Gothons that might be hiding.  It's dead quiet, too quiet."
    print "You stand up and run to the far side of the room and find the"
    print "neutron bomb in its container.  There's a keypad lock on the box"
    print "and you need the code to get the bomb out.  If you get the code"
```

```
    print "wrong 10 times then the lock closes forever and you can't"
    print "get the bomb.  The code is 3 digits."
    code = "%d%d%d" % (randint(1,9), randint(1,9), randint(1,9))
    guess = raw_input("[keypad]> ")
    guesses = 0

    while guess != code and guesses < 10:
        print "BZZZZEDDD!"
        guesses += 1
        guess = raw_input("[keypad]> ")

    if guess == code:
        print "The container clicks open and the seal breaks, letting gas out."
        print "You grab the neutron bomb and run as fast as you can to the"
        print "bridge where you must place it in the right spot."
        return 'the_bridge'
    else:
        print "The lock buzzes one last time and then you hear a sickening"
        print "melting sound as the mechanism is fused together."
        print "You decide to sit there, and finally the Gothons blow up the"
        print "ship from their ship and you die."
        return 'death'


def the_bridge():
    print "You burst onto the Bridge with the neutron destruct bomb"
    print "under your arm and surprise 5 Gothons who are trying to"
    print "take control of the ship.  Each of them has an even uglier"
    print "clown costume than the last.  They haven't pulled their"
    print "weapons out yet, as they see the active bomb under your"
    print "arm and don't want to set it off."

    action = raw_input("> ")

    if action == "throw the bomb":
        print "In a panic you throw the bomb at the group of Gothons"
        print "and make a leap for the door.  Right as you drop it a"
        print "Gothon shoots you right in the back killing you."
        print "As you die you see another Gothon frantically try to disarm"
        print "the bomb. You die knowing they will probably blow up when"
        print "it goes off."
        return 'death'

    elif action == "slowly place the bomb":
        print "You point your blaster at the bomb under your arm"
```

```
        print "and the Gothons put their hands up and start to sweat."
        print "You inch backward to the door, open it, and then carefully"
        print "place the bomb on the floor, pointing your blaster at it."
        print "You then jump back through the door, punch the close button"
        print "and blast the lock so the Gothons can't get out."
        print "Now that the bomb is placed you run to the escape pod to"
        print "get off this tin can."
        return 'escape_pod'
    else:
        print "DOES NOT COMPUTE!"
        return "the_bridge"

def escape_pod():
    print "You rush through the ship desperately trying to make it to"
    print "the escape pod before the whole ship explodes.  It seems like"
    print "hardly any Gothons are on the ship, so your run is clear of"
    print "interference.  You get to the chamber with the escape pods, and"
    print "now need to pick one to take.  Some of them could be damaged"
    print "but you don't have time to look.  There's 5 pods, which one"
    print "do you take?"

    good_pod = randint(1,5)
    guess = raw_input("[pod #]> ")


    if int(guess) != good_pod:
        print "You jump into pod %s and hit the eject button." % guess
        print "The pod escapes out into the void of space, then"
        print "implodes as the hull ruptures, crushing your body"
        print "into jam jelly."
        return 'death'
    else:
        print "You jump into pod %s and hit the eject button." % guess
        print "The pod easily slides out into space heading to"
        print "the planet below.  As it flies to the planet, you look"
        print "back and see your ship implode then explode like a"
        print "bright star, taking out the Gothon ship at the same"
        print "time.  You won!"
        exit(0)


ROOMS = {
    'death': death,
    'central_corridor': central_corridor,
    'laser_weapon_armory': laser_weapon_armory,
```

```
    'the_bridge': the_bridge,
    'escape_pod': escape_pod
}


def runner(map, start):
    next = start

    while True:
        room = map[next]
        print "\n--------"
        next = room()

runner(ROOMS, 'central_corridor')
```

代码不少，不过还是从头写完吧。确认它能运行，然后玩一下看看。

你应该看到的结果

我玩起来时这样的：

```
$ python ex41.py

--------
The Gothons of Planet Percal #25 have invaded your ship and destroyed
your entire crew.  You are the last surviving member and your last
mission is to get the neutron destruct bomb from the Weapons Armory,
put it in the bridge, and blow the ship up after getting into an
escape pod.


You're running down the central corridor to the Weapons Armory when
a Gothon jumps out, red scaly skin, dark grimy teeth, and evil clown costume
flowing around his hate filled body.  He's blocking the door to the
Armory and about to pull a weapon to blast you.
> dodge!
Like a world class boxer you dodge, weave, slip and slide right
as the Gothon's blaster cranks a laser past your head.
In the middle of your artful dodge your foot slips and you
bang your head on the metal wall and pass out.
You wake up shortly after only to die as the Gothon stomps on
your head and eats you.

--------
```

```
Such a luser.
learnpythehardway $ python ex/ex41.py

--------
The Gothons of Planet Percal #25 have invaded your ship and destroyed
your entire crew.  You are the last surviving member and your last
mission is to get the neutron destruct bomb from the Weapons Armory,
put it in the bridge, and blow the ship up after getting into an
escape pod.



You're running down the central corridor to the Weapons Armory when
a Gothon jumps out, red scaly skin, dark grimy teeth, and evil clown costume
flowing around his hate filled body.  He's blocking the door to the
Armory and about to pull a weapon to blast you.
> tell a joke
Lucky for you they made you learn Gothon insults in the academy.
You tell the one Gothon joke you know:
Lbhe zbgure vf fb sng, jura fur fvgf nebhaq gur ubhfr, fur fvgf nebhaq gur ubhfr.
The Gothon stops, tries not to laugh, then busts out laughing and can't move.
While he's laughing you run up and shoot him square in the head
putting him down, then jump through the Weapon Armory door.

--------
You do a dive roll into the Weapon Armory, crouch and scan the room
for more Gothons that might be hiding.  It's dead quiet, too quiet.
You stand up and run to the far side of the room and find the
neutron bomb in its container.  There's a keypad lock on the box
and you need the code to get the bomb out.  If you get the code
wrong 10 times then the lock closes forever and you can't
get the bomb.  The code is 3 digits.
[keypad]> 123
BZZZZEDDD!
[keypad]> 234
BZZZZEDDD!
[keypad]> 345
BZZZZEDDD!
[keypad]> 456
BZZZZEDDD!
[keypad]> 567
BZZZZEDDD!
[keypad]> 678
BZZZZEDDD!
[keypad]> 789
BZZZZEDDD!
```

```
[keypad]> 384
BZZZZEDDD!
[keypad]> 764
BZZZZEDDD!
[keypad]> 354
BZZZZEDDD!
[keypad]> 263
The lock buzzes one last time and then you hear a sickening
melting sound as the mechanism is fused together.
You decide to sit there, and finally the Gothons blow up the
ship from their ship and you die.

--------
You died.  You kinda suck at this.
```

加分习题

1. 解释一下返回至下一个房间的工作原理。
2. 创建更多的房间，让游戏规模变大。
3. 除了让每个函数打印自己以外，再学习一下“文档字符串 (doc strings)”式的注解。看看你能不能将房间描述写成文档注解，然后修改运行它的代码，让它把文档注解打印出来。
4. 一旦你用了文档注解作为房间描述，你还需要让这个函数打印出用户提示吗？试着让运行函数的代码打出用户提示来，然后将用户输入传递到各个函数。你的函数应该只是一些 if 语句组合，将结果打印出来，并且返回下一个房间。
5. 这其实是一个小版本的“有限状态机 (finite state machine)”，找资料阅读了解一下，虽然你可能看不懂，但还是找来看看吧。

习题 42：物以类聚

虽说将函数放到字典里是很有趣的一件事情，你应该也会想到“如果 Python 能自动为你做这件事情该多好”。事实上也的确有，那就是 `class` 这个关键字。你可以使用 `class` 创建更棒的“函数字典”，比你在上节练习中做的强大多了。Class（类）有着各种各样强大的功能和用法，但本书不会深入讲这些内容，在这里，你只要你学会把它们当作高级的“函数字典”使用就可以了。

用到“class”的编程语言被称作“Object Oriented Programming（面向对象编程）”语言。这是一种传统的编程方式，你需要做出“东西”来，然后你“告诉”这些东西去完成它们的工作。类似的事情你其实已经做过不少了，只不过还没有意识到而已。记得你做过的这个吧：

```
stuff = ['Test', 'This', 'Out']
print ' '.join(stuff)
```

其实你这里已经使用了 class。“stuff“ 这个变量其实是一个 `list class` （列表类）。而 `' '.join(stuff)` 里调用函数 `join` 的字符串 `' '` （就是一个空格）也是一个 class ——它是一个 string class (字符串类)。到处都是 class！

还有一个概念是 object（物件），不过我们暂且不提。当你创建过几个 class 后就会学到了。你怎样创建 class 呢？和你创建 `ROOMS` 的方法差不多，但其实更简单：

```
class TheThing(object):

    def __init__(self):
        self.number = 0

    def some_function(self):
        print "I got called."

    def add_me_up(self, more):
        self.number += more
        return self.number

# two different things
a = TheThing()
b = TheThing()
```

```
a.some_function()
b.some_function()

print a.add_me_up(20)
print a.add_me_up(20)
print b.add_me_up(30)
print b.add_me_up(30)

print a.number
print b.number
```

> **Warning:** 嗯，你开始看到 Python 的“疣子”了。Python 是一门比较旧的语言，其中包含很多丑陋的设计决定。为了将这些丑陋设计掩盖过去，他们就做了一些新的丑陋设计，然后告诉人们让他们习惯这些新的坏设计。“class TheThing(object)“ 就是其中一个例子。这里我就不展开讲了，不过你也不必操心为什么你的 class 要在后面添一个 `(object)`，只要跟着这样做就可以了，否则将来总有一天别的 Python 程序员会吼你让你这样做。后面我们再讲为什么。

你看到参数里的 `self` 了吧？你知道它是什么东西吗？对了，它就是 Python 创建的额外的一个参数，有了它你才能实现 `a.some_function()`` 这种用法，这时它就会把 \ 前者翻译成 ``some_function(a)` 执行下去。为什么用 `self` 呢？因为你的函数并不知道你的这个“实例”是来自叫 `TheThing` 或者别的名字的 class。所以你只要使用一个通用的名字 `self`。这样你写出来的函数就会在任何情况下都能正常工作。

其实你可以使用 `self` 以外的别的字眼，不过如果你这样做的话，你就会成为所有 Python 程序员的众之矢的，所以还是随大流的好。只有变态才会在这里乱改，我教你的没错。对以后会读到你的代码的人好点儿，因为你现在的代码 10 年以后所有的代码都会是一团糟。

接下来，看到 `__init__` 函数了吗？这就是你为 Python class 设置内部变量的方式。你可以使用 `.` 将它们设置到 `self` 上面。另外看到我们使用了 `add_me_up()` 为你创建的 `self.number` 加值。后面你可以看到我们怎样可以使用这种方法为数字加值，然后打印出来。

Class 是很强大的东西，你应该好好读读相关的东西。尽可能多找些东西读并且多多实验。你其实知道它们该怎么用，只要试试就知道了。其实我马上就要去练吉他了，所以我不会让你写练习了。你将使用 class 写一个练习。

接下来我们将把习题 41 的内容重写，不过这回我们将使用 class：

```
from sys import exit
from random import randint

class Game(object):

    def __init__(self, start):
        self.quips = [
            "You died.  You kinda suck at this.",
             "Your mom would be proud. If she were smarter.",
             "Such a luser.",
```

```
            "I have a small puppy that's better at this."
        ]
        self.start = start

    def play(self):
        next = self.start

        while True:
            print "\n--------"
            room = getattr(self, next)
            next = room()


    def death(self):
        print self.quips[randint(0, len(self.quips)-1)]
        exit(1)

    def central_corridor(self):
        print "The Gothons of Planet Percal #25 have invaded your ship and destroyed"
        print "your entire crew.  You are the last surviving member and your last"
        print "mission is to get the neutron destruct bomb from the Weapons Armory,"
        print "put it in the bridge, and blow the ship up after getting into an "
        print "escape pod."
        print "\n"
        print "You're running down the central corridor to the Weapons Armory when"
        print "a Gothon jumps out, red scaly skin, dark grimy teeth, and evil clown costume"
        print "flowing around his hate filled body.  He's blocking the door to the"
        print "Armory and about to pull a weapon to blast you."

        action = raw_input("> ")

        if action == "shoot!":
            print "Quick on the draw you yank out your blaster and fire it at the Gothon."
            print "His clown costume is flowing and moving around his body, which throws"
            print "off your aim.  Your laser hits his costume but misses him entirely.  This"
            print "completely ruins his brand new costume his mother bought him, which"
            print "makes him fly into an insane rage and blast you repeatedly in the face until"
            print "you are dead.  Then he eats you."
            return 'death'

        elif action == "dodge!":
            print "Like a world class boxer you dodge, weave, slip and slide right"
            print "as the Gothon's blaster cranks a laser past your head."
            print "In the middle of your artful dodge your foot slips and you"
            print "bang your head on the metal wall and pass out."
```

```
            print "You wake up shortly after only to die as the Gothon stomps on"
            print "your head and eats you."
            return 'death'

        elif action == "tell a joke":
            print "Lucky for you they made you learn Gothon insults in the academy."
            print "You tell the one Gothon joke you know:"
            print "Lbhe zbgure vf fb sng, jura fur fvgf nebhaq gur ubhfr, fur fvgf nebhaq gur ubhfr."
            print "The Gothon stops, tries not to laugh, then busts out laughing and can't move."
            print "While he's laughing you run up and shoot him square in the head"
            print "putting him down, then jump through the Weapon Armory door."
            return 'laser_weapon_armory'

        else:
            print "DOES NOT COMPUTE!"
            return 'central_corridor'

    def laser_weapon_armory(self):
        print "You do a dive roll into the Weapon Armory, crouch and scan the room"
        print "for more Gothons that might be hiding.  It's dead quiet, too quiet."
        print "You stand up and run to the far side of the room and find the"
        print "neutron bomb in its container.  There's a keypad lock on the box"
        print "and you need the code to get the bomb out.  If you get the code"
        print "wrong 10 times then the lock closes forever and you can't"
        print "get the bomb.  The code is 3 digits."
        code = "%d%d%d" % (randint(1,9), randint(1,9), randint(1,9))
        guess = raw_input("[keypad]> ")
        guesses = 0

        while guess != code and guesses < 10:
            print "BZZZZEDDD!"
            guesses += 1
            guess = raw_input("[keypad]> ")

        if guess == code:
            print "The container clicks open and the seal breaks, letting gas out."
            print "You grab the neutron bomb and run as fast as you can to the"
            print "bridge where you must place it in the right spot."
            return 'the_bridge'
        else:
            print "The lock buzzes one last time and then you hear a sickening"
            print "melting sound as the mechanism is fused together."
            print "You decide to sit there, and finally the Gothons blow up the"
            print "ship from their ship and you die."
            return 'death'
```

```


    def the_bridge(self):
        print "You burst onto the Bridge with the netron destruct bomb"
        print "under your arm and surprise 5 Gothons who are trying to"
        print "take control of the ship.  Each of them has an even uglier"
        print "clown costume than the last.  They haven't pulled their"
        print "weapons out yet, as they see the active bomb under your"
        print "arm and don't want to set it off."

        action = raw_input("> ")

        if action == "throw the bomb":
            print "In a panic you throw the bomb at the group of Gothons"
            print "and make a leap for the door.  Right as you drop it a"
            print "Gothon shoots you right in the back killing you."
            print "As you die you see another Gothon frantically try to disarm"
            print "the bomb. You die knowing they will probably blow up when"
            print "it goes off."
            return 'death'

        elif action == "slowly place the bomb":
            print "You point your blaster at the bomb under your arm"
            print "and the Gothons put their hands up and start to sweat."
            print "You inch backward to the door, open it, and then carefully"
            print "place the bomb on the floor, pointing your blaster at it."
            print "You then jump back through the door, punch the close button"
            print "and blast the lock so the Gothons can't get out."
            print "Now that the bomb is placed you run to the escape pod to"
            print "get off this tin can."
            return 'escape_pod'
        else:
            print "DOES NOT COMPUTE!"
            return "the_bridge"

    def escape_pod(self):
        print "You rush through the ship desperately trying to make it to"
        print "the escape pod before the whole ship explodes.  It seems like"
        print "hardly any Gothons are on the ship, so your run is clear of"
        print "interference.  You get to the chamber with the escape pods, and"
        print "now need to pick one to take.  Some of them could be damaged"
        print "but you don't have time to look.  There's 5 pods, which one"
        print "do you take?"

        good_pod = randint(1,5)
```

```
        guess = raw_input("[pod #]> ")


        if int(guess) != good_pod:
            print "You jump into pod %s and hit the eject button." % guess
            print "The pod escapes out into the void of space, then"
            print "implodes as the hull ruptures, crushing your body"
            print "into jam jelly."
            return 'death'
        else:
            print "You jump into pod %s and hit the eject button." % guess
            print "The pod easily slides out into space heading to"
            print "the planet below.  As it flies to the planet, you look"
            print "back and see your ship implode then explode like a"
            print "bright star, taking out the Gothon ship at the same"
            print "time.  You won!"
            exit(0)


a_game = Game("central_corridor")
a_game.play()
```

你应该看到的结果

这个版本的游戏和你的上一版效果应该是一样的，其实有些代码都几乎一样。比较一下两版代码，弄懂其中不同的地方，重点需要理解这些东西：

1. 怎样创建一个 `class Game(object)` 并且放函数到里边去。
2. `__init__` 是一个特殊的初始方法，可以预设重要的变量在里边。
3. 为 class 添加函数的方法是将函数在 class 下再缩进一阶，class 的架构就是通过缩进实现的，这点很重要。
4. 你在函数里的内容又缩进了一阶。
5. 注意冒号的用法。
6. 理解 `self` 的概念，以及它在 `__init__` 、`play` 、`death` 里是怎样使用的。
7. 研究 `play` 里的 `getattr` 的功能，这样你就能明白 `play` 所做的事情。其实你可以手动在 Python 命令行实验一下，从而弄懂它。
8. 最后我们怎样创建了一个 `Game` ，然后通过 `play()` 让所有的东西运行起来。

加分习题

1. 研究一下 `__dict__` 是什么东西，应该怎样使用。
2. 再为游戏添加一些房间，确认自己已经学会使用 class 。
3. 创建一个新版本，里边使用两个 class，其中一个是 `Map` ，另一个是 `Engine` 。提示: 把 `play` 放到 `Engine` 里面。

习题 43：你来制作一个游戏

你要开始学会自食其力了。通过阅读这本书你应该已经学到了一点，那就是你需要的所有的信息网上都有，你只要去搜索就能找到。唯一困扰你的就是如何使用正确的词汇进行搜索。学到现在，你在挑选搜索关键字方面应该已经有些感觉了。现在已经是时候了，你需要尝试写一个大的项目，并让它运行起来。

以下是你的需求：

1. 制作一个截然不同的游戏。
2. 使用多个文件，并使用 `import` 调用这些文件。确认自己知道 `import` 的用法。
3. 对于每个房间使用一个 class，class 的命名要能体现出它的用处。例如 `GoldRoom`、`KoiPondRoom`。
4. 你的执行器代码应该了解这些房间，所以创建一个 class 来调用并且记录这些房间。有很多种方法可以达到这个目的，不过你可以考虑让每个房间返回下一个房间，或者设置一个变量，让它指定下一个房间是什么。

其他的事情就全靠你了。花一个星期完成这件任务，做一个你能做出来的最好的游戏。使用你学过的任何东西（类，函数，字典，列表……）来改进你的程序。这节课的目的是教你如何构建 class 出来，而这些 class 又能调用到其它 Python 文件中的 class。

我不会详细地告诉你告诉你怎样做，你需要自己完成。试着下手吧，编程就是解决问题的过程，这就意味着你要尝试各种可能性，进行实验，经历失败，然后丢掉你做出来的东西重头开始。当你被某个问题卡住的时候，你可以向别人寻求帮助，并把你的代码贴出来给他们看。如果有人刻薄你，别理他们，你只要集中精力在帮你的人身上就可以了。持续修改和清理你的代码，直到它完整可执行为止，然后再研究一下看它还能不能被改进。

祝你好运，下个星期你做出游戏后我们再见。

习题 44：给你的游戏打分

这节练习的目的是检查评估你的游戏。也许你只完成了一半，卡在那里没有进行下去，也许你勉强做出来了。不管怎样，我们将串一下你应该弄懂的一些东西，并确认你的游戏里有使用到它们。我们将学习如何用正确的格式构建 class，使用 class 的一些通用习惯，另外还有很多的“书本知识”让你学习。

为什么我会让你先行尝试，然后才告诉你正确的做法呢？因为从现在开始你要学会“自给自足”，以前是我牵着你前行，以后就得靠你自己了。后面的习题我只会告诉你你的任务，你需要自己去完成，在你完成后我再告诉你如何可以改进你的作业。

一开始你会觉得很困难并且很不习惯，但只要坚持下去，你就会培养出自己解决问题的能力。你还会找出创新的方法解决问题，这比从课本中拷贝解决方案强多了。

函数的风格

以前我教过的怎样写好函数的方法一样是适用的，不过这里要添加几条：

- 由于各种各样的原因，程序员将 class (类) 里边的函数称作 `method` （方法）。很大程度上这只是个市场策略（用来推销 OOP），不过如果你把它们称作“函数”的话，是会有啰嗦的人跳出来纠正你的。如果你觉得他们太烦了，你可以告诉他们从数学方面演示一下“函数”和“方法”究竟有什么不同，这样他们会很快闭嘴的。
- 在你使用 class 的过程中，很大一部分时间是告诉你的 class 如何“做事情”。给这些函数命名的时候，与其命名成一个名词，不如命名为一个动词，作为给 class 的一个命令。就和 list 的 `pop` (抛出) 函数一样，它相当于说：“嘿，列表，把这东西给我 pop 出去。”它的名字不是 `remove_from_end_of_list` ，因为即使它的功能的确是这样，这一串字符也不是一个命令。
- 让你的函数保持简单小巧。由于某些原因，有些人开始学习 class 后就会忘了这一条。

类的风格

- 你的 class 应该使用“camel case（驼峰式大小写）”，例如你应该使用 `SuperGoldFactory` 而不是 `super_gold_factory`。
- 你的 `__init__` 不应该做太多的事情，这会让 class 变得难以使用。

- 你的其它函数应该使用“underscore format（下划线隔词）”，所以你可以写 `my_awesome_hair`，而不是 `myawesomehair` 或者 `MyAwesomeHair` 。
- 用一致的方式组织函数的参数。如果你的 class 需要处理 users、dogs、和 cats，就保持这个次序（特别情况除外）。如果一个函数的参数是 `(dog, cat, user)` ，另一个的是 `(user, cat, dog)` ，这会让函数使用起来很困难。
- 不要对全局变量或者来自模组的变量进行重定义或者赋值，让这些东西自顾自就行了。
- 不要一根筋式地维持风格一致性，这是思维力底下的妖怪喽啰做的事情。一致性是好事情，不过愚蠢地跟着别人遵从一些白痴口号是错误的行为——这本身就是一种坏的风格。好好为自己照想把。
- 永远永远都使用 `class Name(object)` 的方式定义 class，否则你会碰到大麻烦。

代码风格

- 为了以方便他人阅读，为自己的代码字符之间留下一些空白。你将会看到一些很差的程序员，他们写的代码还算通顺，但字符之间没有任何空间。这种风格在任何编程语言中都是坏习惯，人的眼睛和大脑会通过空白和垂直对齐的位置来扫描和区隔视觉元素，如果你的代码里没有任何空白，这相当于为你的代码上了迷彩装。如果一段代码你无法朗读出来，那么这段代码的可读性可能就有问题。如你找不到让某个东西易用的方法，试着也朗读出来。这样不仅会逼迫你慢速而且真正仔细阅读过去，还会帮你找到难读的段落，从而知道那些代码的易读性需要作出改进。
- 学着模仿别人的风格写 Python 程序，直到哪天你找到你自己的风格为止。
- 一旦你有了自己的风格，也别把它太当回事。程序员工作的一部分就是和别人的代码打交道，有的人审美就是很差。相信我，你的审美某一方面一定也很差，只是你从未意识到而已。
- 如果你发现有人写的代码风格你很喜欢，那就模仿他们的风格。

好的注释

- 有程序员会告诉你，说你的代码需要有足够的可读性，这样你就无需写注释了。他们会以自己接近官腔的声音说“所以你永远都不应该写代码注释。”这些人要么是一些顾问型的人物，如果别人无法使用他们的代码，就会付更多钱给他们让他们解决问题。要么他们能力不足，从来没有跟别人合作过。别理会这些人，好好写你的注解。
- 写注解的时候，描述清楚为什么你要这样做。代码只会告诉你“这样实现”，而不会告诉你“为什么要这样实现”，而后者比前者更重要。
- 当你为函数写文档注解的时候，记得为别的代码使用者也写些东西。你不需要狂写一大堆，但一两句话谢谢这个函数的用法还是很有用的。
- 最后要说的是，虽然注解是好东西，太多的注解就不见得是了。而且注解也是需要维护的，你要尽量让注解短小精悍一语中的，如果你对代码做了更改，记得检查并更新相关的注解，确认它们还是正确的。

为你的游戏评分

现在我要求你假装成是我，板起脸来，把你的代码打印出来，然后拿一支红笔，把代码中所有的错误都标出来。你要充分利用你在本章以及前面学到的知识。等你批改完了，我要求你把所有的错误改对。这个过程我需要你多重复几次，争取找到更多的可以改进的地方。使用我前面教过的方法，把代码分解成最细小的单元一一进行分析。

这节练习的目的是训练你对于细节的关注程度。等你检查完自己的代码，再找一段别人的代码用这种方法检查一遍。把代码打印出来，检查出所有代码和风格方面的错误，然后试着在不改坏别人代码的前提下把它们修改正确、

这周我要求你的事情就是批改和纠错，包含你自己的代码和别人的代码，再没有别的了。这节习题难度还是挺大，不过一旦你完成了任务，你学过的东西就会牢牢记在脑中。

习题 45：对象、类、以及从属关系

有一个重要的概念你需要弄明白，那就是“类 (class)”和“对象 (object)”的区别。问题在于，class 和 object 并没有真正的不同。它们其实是同样的东西，只是在不同的时间名字不同罢了。我用禅语来解释一下吧：

鱼和泥鳅有什么区别？

这个问题有没有让你有点晕呢？说真的，坐下来想一分钟。我的意思是说，鱼和泥鳅是不一样，不过它们其实也是一样的是不是？泥鳅是鱼的一种，所以说没什么不同，不过泥鳅又有些特别，它和别的种类的鱼的确不一样，比如泥鳅和黄鳝就不一样。所以泥鳅和鱼既相同又不同。怪了。

这个问题让人晕的原因是大部分人不会这样去思考问题，其实每个人都懂这一点，你无须去思考鱼和泥鳅的区别，因为你知道它们之间的关系。你知道泥鳅是鱼的一种，而且鱼还有别的种类，根本就没必要去思考这类问题。

让我们更进一步，假设你有一只水桶，里边有三条泥鳅。假设你的好人卡多到没地方用，于是你给它们分别取名叫小方，小斌，小星。现在想想这个问题：

小方和泥鳅有什么区别？

这个问题一样的奇怪，但比起鱼和泥鳅的问题来还好点。你知道小方是一条泥鳅，所以他并没什么不同，他只是泥鳅的一个“实例 (instance)”。小斌和小星一样也是泥鳅的实例。我的意思是说，它们是由泥鳅创建出来的，而且代表着和泥鳅一样的属性。

所以我们的思维方式是（你可能会有点不习惯）：鱼是一个“类 (class)”，泥鳅是一个“类 (class)”，而小方是一个“对象 (object)”。仔细想想，然后我再一点一点慢慢解释给你。

鱼是一个“类”，表示它不是一个真正的东西，而是一个用来描述具有同类属性的实例的概括性词汇。你有鳍？你有鳔？你住在水里？好吧那你就是一条鱼。

后来河蟹养殖专家路过，看到你的水桶，于是告诉你：“小伙子，你这些鱼是泥鳅。”专家一出，真相即现。并且专家还定义了一个新的叫做“泥鳅”的“类”，而这个“类”又有它特定的属性。细长条？有胡须？爱钻泥巴？吃起来味道还可以？那你就是一条泥鳅。

最后家庭煮父过来了，他跟河蟹专家说：“非也非也，你看到的是泥鳅，我看到的是小方，而且我要把小方和剁椒配一起做一道小菜。”于是你就有了一只叫做小方的泥鳅的“实例 (instance)”（泥鳅也是鱼的一个“实例”），并且你使用了它（把它塞到你的胃里了），这样它就是一个“对象 (object)”。

这会你应该了解了：小方是泥鳅的成员，而泥鳅又是鱼的成员。这里的关系式：对象属于某个类，而某个类又属于另一个类。

写成代码是什么样子

这个概念有点绕人，不过实话说，你只要在创建和使用 class 的时候操心一下就可以了。我来给你两个区分 `Class` 和 `Object` 的小技巧。

首先针对类和对象，你需要学会两个说法，"is-a(是啥)"和"has-a(有啥)"。"是啥"要用在谈论"两者以类的关系互相关联"的时候，而"有啥"要用在"两者无共同点，仅是互为参照"的时候。

接下来，通读这段代码，将每一个注解为 `##??` 的位置标明他是"is-a"还是"has-a"的关系，并讲明白这个关系是什么。在代码的开始我还举了几个例子，所以你只要写剩下的就可以了。

记住，"是啥"指的是鱼和泥鳅的关系，而"有啥"指的是泥鳅和鳃的关系。

（译注：为了解释方便，译文使用了中文鱼名。原文使用的是"三文鱼 (salmon)"和"大比目鱼 (halibut)"，名字也是英文常用人名。）

```python
## Animal is-a object (yes, sort of confusing) look at the extra credit
class Animal(object):
    pass

## ??
class Dog(Animal):

    def __init__(self, name):
        ## ??
        self.name = name

## ??
class Cat(Animal):

    def __init__(self, name):
        ## ??
        self.name = name

## ??
class Person(object):

    def __init__(self, name):
        ## ??
        self.name = name

        ## Person has-a pet of some kind
        self.pet = None
```

```python

## ??
class Employee(Person):

    def __init__(self, name, salary):
        ## ?? hmm what is this strange magic?
        super(Employee, self).__init__(name)
        ## ??
        self.salary = salary

## ??
class Fish(object):
    pass

## ??
class Salmon(Fish):
    pass

## ??
class Halibut(Fish):
    pass


## rover is-a Dog
rover = Dog("Rover")

## ??
satan = Cat("Satan")

## ??
mary = Person("Mary")

## ??
mary.pet = satan

## ??
frank = Employee("Frank", 120000)

## ??
frank.pet = rover

## ??
flipper = Fish()

## ??
```

```
crouse = Salmon()

## ??
harry = Halibut()
```

关于 class Name(object)

记得我曾经强迫让你使用 `class Name(object)` 却没告诉你为什么吧，现在你已经知道了“类”和“对象”的区别，我就可以告诉你原因了。如果我早告诉你的话，你可能会晕掉，也学不会这门技术了。

真正的原因是在 Python 早期，它对于 `class` 的定义在很多方面都是严重有问题的。当他们承认这一点的时候已经太迟了，所以逼不得已，他们需要支持这种有问题的 `class`。为了解决已有的问题，他们需要引入一种“新类”，这样的话“旧类”还能继续使用，而你也有一个新的正确的类可以使用了。

这就用到了“类即是对象”的概念。他们决定用小写的“object”这个词作为一个类，让你在创建新类时从它继承下来。有点晕了吧？一个类从另一个类继承，而后者虽然是个类，但名字却叫“object”……不过在定义类的时候，别忘记要从 object 继承就好了。

的确如此。一个词的不同就让这个概念变得更难理解，让我不得不现在才讲给你。现在你可以试着去理解“一个是对象的类”这个概念了，如果你感兴趣的话。

不过我还是建议你别去理解了，干脆完全忘记旧格式和新格式类的区别吧，就假设 Python 的 class 永远都要求你加上 (object) 好了，你的脑力要留着思考更重要的问题。

加分习题

1. 研究一下为什么 Python 添加了这个奇怪的叫做 `object` 的 class，它究竟有什么含义呢？
2. 有没有办法把 `Class` 当作 `Object` 使用呢？
3. 在习题中为 animals、fish、还有 people 添加一些函数，让它们做一些事情。看看当函数在 Animal 这样的“基类 (base class)”里和在 Dog 里有什么区别。
4. 找些别人的代码，理清里边的“是啥”和“有啥”的关系。
5. 使用列表和字典创建一些新的一对应多的“has-many”的关系。
6. 你认为会有一种“has-many”的关系吗？阅读一下关于“多重继承 (multiple inheritance)”的资料，然后尽量避免这种用法。

习题 46：一个项目骨架

这里你将学会如何建立一个项目“骨架”目录。这个骨架目录具备让项目跑起来的所有基本内容。它里边会包含你的项目文件布局、自动化测试代码，模组，以及安装脚本。当你建立一个新项目的时候，只要把这个目录复制过去，改改目录的名字，再编辑里边的文件就行了。

骨架内容

首先使用下述命令创建你的骨架目录：

```
~ $ mkdir -p projects
~ $ cd projects/
~/projects $ mkdir skeleton
~/projects $ cd skeleton
~/projects/skeleton $ mkdir bin NAME tests docs
```

我使用了一个叫 projects 的目录，用来存放我自己的各个项目。然后我在里边建立了一个叫做 skeleton 的文件夹，这就是我们新项目的基础目录。其中叫做 NAME 的文件夹是你的项目的主文件夹，你可以将它任意取名。

接下来我们要配置一些初始文件：

```
~/projects/skeleton $ touch NAME/__init__.py
~/projects/skeleton $ touch tests/__init__.py
```

以上命令为你创建了空的模组目录，以供你后面为其添加代码。然后我们需要建立一个 setup.py 文件，这个文件在安装项目的时候我们会用到它：

```
try:
    from setuptools import setup
except ImportError:
    from distutils.core import setup

config = {
    'description': 'My Project',
    'author': 'My Name',
    'url': 'URL to get it at.',
```

```
    'download_url': 'Where to download it.',
    'author_email': 'My email.',
    'version': '0.1',
    'install_requires': ['nose'],
    'packages': ['NAME'],
    'scripts': [],
    'name': 'projectname'
}

setup(**config)
```

编辑这个文件，把自己的联系方式写进去，然后放到那里就行了。

最后你需要一个简单的测试专用的骨架文件叫 `tests/NAME_tests.py`:

```
from nose.tools import *
import NAME

def setup():
    print "SETUP!"

def teardown():
    print "TEAR DOWN!"

def test_basic():
    print "I RAN!"
```

Python 软件包的安装

你需要预先安装一些软件包，不过问题就来了。我的本意是让这本书越清晰越干净越好，不过安装软件的方法是在是太多了，如果我要一步一步写下来，那 10 页都写不完，而且告诉你吧，我本来就是个懒人。

所以我不会提供详细的安装步骤了，我只会告诉你需要安装哪些东西，然后让你自己搞定。这对你也有好处，因为你将打开一个全新的世界，里边充满了其他人发布的 Python 软件。

接下来你需要安装下面的软件包：

1. pip – http://pypi.python.org/pypi/pip
2. distribute – http://pypi.python.org/pypi/distribute
3. nose – http://pypi.python.org/pypi/nose/
4. virtualenv – http://pypi.python.org/pypi/virtualenv

不要只是手动下载并且安装这些软件包，你应该看一下别人的建议，尤其看看针对你的操作系统别人是怎样建议你安装和使用的。同样的软件包在不一样的操作系统上面的安装方式是不一样的，不一样版本的 Linux 和 OSX 会有不同，而 Windows 更是不同。

我要预先警告你，这个过程会是相当无趣。在业内我们将这种事情叫做 “yak shaving(剃牦牛)”。它指的是在你做一件有意义的事情之前的一些准备工作，而这些准备工作又是及其无聊冗繁的。你要做一个很酷的 Python 项目，但是创建骨架目录需要你安装一些软件包，而安装软件包之前你还要安装 package installer (软件包安装工具)，而要安装这个工具你还得先学会如何在你的操作系统下安装软件，真是烦不胜烦呀。

无论如何，还是克服困难把。你就把它当做进入编程俱乐部的一个考验。每个程序员都会经历这条道路，在每一段“酷”的背后总会有一段“烦”的。

测试你的配置

安装了所有上面的软件包以后，你就可以做下面的事情了：

```
~/projects/skeleton $ nosetests
.
----------------------------------------------------------------------
Ran 1 test in 0.007s

OK
```

下一节练习中我会告诉你 `nosetests` 的功能，不过如果你没有看到上面的画面，那就说明你哪里出错了。确认一下你的 `NAME` 和 `tests` 目录下存在 `__init__.py`，并且你没有把 `tests/NAME_tests.py` 命名错。

使用这个骨架

剃牦牛的事情已经做的差不多了，以后每次你要新建一个项目时，只要做下面的事情就可以了：

1. 拷贝这份骨架目录，把名字改成你新项目的名字。
2. 再将 NAME 模组更名为你需要的名字，它可以是你项目的名字，当然别的名字也行。
3. 编辑 setup.py 让它包含你新项目的相关信息。
4. 重命名 `tests/NAME_tests.py` ，让它的名字匹配到你模组的名字。
5. 使用 `nosetests` 检查有无错误。
6. 开始写代码吧。

小测验

这节练习没有加分习题，不过需要你做一个小测验：

1. 找文档阅读，学会使用你前面安装了的软件包。
2. 阅读关于 `setup.py` 的文档，看它里边可以做多少配置。Python 的安装器并不是一个好软件，所以使用起来也非常奇怪。

3. 创建一个项目，在模组目录里写一些代码，并让这个模组可以运行。
4. 在 `bin` 目录下放一个可以运行的脚本，找材料学习一下怎样创建可以在系统下运行的 Python 脚本。
5. 在你的 `setup.py` 中加入 `bin` 这个目录，这样你安装时就可以连它安装进去。
6. 使用 `setup.py` 安装你的模组，并确定安装的模组可以正常使用，最后使用 `pip` 将其卸载。

练习 47：自动化测试

为了确认游戏的功能是否正常，你需要一遍一遍地在你的游戏中输入命令。这个过程是很枯燥无味的。如果能写一小段代码用来测试你的代码岂不是更好？然后只要你对程序做了任何修改，或者添加了什么新东西，你只要“跑一下你的测试”，而这些测试能确认程序依然能正确运行。这些自动测试不会抓到所有的 bug，但可以让你无需重复输入命令运行你的代码，从而为你节约很多时间。

从这一章开始，以后的练习将不会有“你应该看到的结果”这一节，取而代之的是一个“你应该测试的东西”一节。从现在开始，你需要为自己写的所有代码写自动化测试，而这将让你成为一个更好的程序员。

我不会试图解释为什么你需要写自动化测试。我要告诉你的是，你想要成为一个程序员，而程序的作用是让无聊冗繁的工作自动化，测试软件毫无疑问是无聊冗繁的，所以你还是写点代码让它为你测试的更好。

这应该是你需要的所有的解释了。因为你写单元测试的原因是让你的大脑更加强健。你读了这本书，写了很多代码让它们实现一些事情。现在你将更进一步，写出懂得你写的其他代码的代码。这个写代码测试你写的其他代码的过程将强迫你清楚的理解你之前写的代码。这会让你更清晰地了解你写的代码实现的功能及其原理，而且让你对细节的注意更上一个台阶。

撰写测试用例

我们将拿一段非常简单的代码为例，写一个简单的测试，这个测试将建立在上节我们创建的项目骨架上面。

首先从你的项目骨架创建一个叫做 ex47 的项目。确认该改名称的地方都有改过，尤其是 tests/ex47_tests.py 这处不要写错，另外运行 nosetest 确认一下没有错误信息。检查一下 tests/skel_tests.pyc 这个文件，有的话就把它删掉，这一点需要尤其注意。

接下来创建一个简单的 ex47/game.py 文件，里边放一些用来被测试的代码。我们现在放一个傻乎乎的小 class 进去，用来作为我们的测试对象：

```
class Room(object):

    def __init__(self, name, description):
        self.name = name
        self.description = description
        self.paths = {}

```

```python
    def go(self, direction):
        return self.paths.get(direction, None)

    def add_paths(self, paths):
        self.paths.update(paths)
```

准备好了这个文件，接下来把测试骨架改成这样子：

```python
from nose.tools import *
from ex47.game import Room


def test_room():
    gold = Room("GoldRoom",
                """This room has gold in it you can grab. There's a
                door to the north.""")
    assert_equal(gold.name, "GoldRoom")
    assert_equal(gold.paths, {})

def test_room_paths():
    center = Room("Center", "Test room in the center.")
    north = Room("North", "Test room in the north.")
    south = Room("South", "Test room in the south.")

    center.add_paths({'north': north, 'south': south})
    assert_equal(center.go('north'), north)
    assert_equal(center.go('south'), south)

def test_map():
    start = Room("Start", "You can go west and down a hole.")
    west = Room("Trees", "There are trees here, you can go east.")
    down = Room("Dungeon", "It's dark down here, you can go up.")

    start.add_paths({'west': west, 'down': down})
    west.add_paths({'east': start})
    down.add_paths({'up': start})

    assert_equal(start.go('west'), west)
    assert_equal(start.go('west').go('east'), start)
    assert_equal(start.go('down').go('up'), start)
```

这个文件 import 了你在 `ex47.game` 创建的 `Room` 这个类，接下来我们要做的就是测试它。于是我们看到一系列的以 `test_` 开头的测试函数，它们就是所谓的“测试用例 (test case)”，每一个测试用例里面都有一小段代码，它们会创建一个或者一些房间，然后去确认房间的功能和你期望的是否一样。它测试了基本的房间功能，然后测试了路径，最后测试了整个地图。

这里最重要的函数时 `assert_equal`，它保证了你设置的变量，以及你在 `Room` 里设置的路径和你的期望相符。如果你得到错误的结果的话，`nosetests` 将会打印出一个错误信息，这样你就可以找到出错的地方并且修正过来。

测试指南

在写测试代码时，你可以照着下面这些不是很严格的指南来做：

1. 测试脚本要放到 `tests/` 目录下，并且命名为 `BLAH_tests.py` ，否则 `nosetests` 就不会执行你的测试脚本了。这样做还有一个好处就是防止测试代码和别的代码互相混掉。
2. 为你的每一个模组写一个测试。
3. 测试用例（函数）保持简短，但如果看上去不怎么整洁也没关系，测试用例一般都有点乱。
4. 就算测试用例有些乱，也要试着让他们保持整洁，把里边重复的代码删掉。创建一些辅助函数来避免重复的代码。当你下次在改完代码需要改测试的时候，你会感谢我这一条建议的。重复的代码会让修改测试变得很难操作。
5. 最后一条是别太把测试当做一回事。有时候，更好的方法是把代码和测试全部删掉，然后重新设计代码。

你应该看到的结果

```
~/projects/simplegame $ nosetests
...
----------------------------------------------------------------------
Ran 3 tests in 0.007s

OK
```

如果一切工作正常的话，你看到的结果应该就是这样。试着把代码改错几个地方，然后看错误信息会是什么，再把代码改正确。

加分习题

1. 仔细读读 nosetest 相关的文档，再去了解一下其他的替代方案。
2. 了解一下 Python 的 “doc tests” ，看看你是不是更喜欢这种测试方式。
3. 改进你游戏里的 Room，然后用它重建你的游戏，这次重写，你需要一边写代码，一边把单元测试写出来。

习题 48: 更复杂的用户输入

你的游戏可能一路跑得很爽，不过你处理用户输入的方式肯定让你不胜其烦了。每一个房间都需要一套自己的语句，而且只有用户完全输入正确后才能执行。你需要一个设备，它可以允许用户以各种方式输入语汇。例如下面的机种表述都应该被支持才对：

- open door
- open the door
- go THROUGH the door
- punch bear
- Punch The Bear in the FACE

也就是说，如果用户的输入和常用英语很接近也应该是可以的，而你的游戏要识别出它们的意思。为了达到这个目的，我们将写一个模组专门做这件事情。这个模组里边会有若干个类，它们互相配合，接受用户输入，并且将用户输入转换成你的游戏可以识别的命令。

英语的简单格式是这个样子的：

- 单词由空格隔开。
- 句子由单词组成。
- 语法控制句子的含义。

所以最好的开始方式是先搞定如何得到用户输入的词汇，并且判断出它们是什么。

我们的游戏语汇

我在游戏里创建了下面这些语汇：

- 表示方向: north, south, east, west, down, up, left, right, back.
- 动词: go, stop, kill, eat.
- 修饰词: the, in, of, from, at, it
- 名词: door, bear, princess, cabinet.
- 数词: 由 0-9 构成的数字。

说到名词，我们会碰到一个小问题，那就是不一样的房间会用到不一样的一组名词，不过让我们先挑一小组出来写程序，以后再做改进把。

如何断句

我们已经有了词汇表，为了分析句子的意思，接下来我们需要找到一个断句的方法。我们对于句子的定义是“空格隔开的单词”，所以只要这样就可以了：

```
stuff = raw_input('> ')
words = stuff.split()
```

目前做到这样就可以了，不过这招在相当一段时间内都不会有问题。

语汇元组

一旦我们知道了如何将句子转化成词汇列表，剩下的就是逐一检查这些词汇，看它们是什么类型。为了达到这个目的，我们将用到一个非常好使的 Python 数据结构，叫做” 元组 (tuple)”。元组其实就是一个不能修改的列表。创建它的方法和创建列表差不多，成员之间需要用逗号隔开，不过方括号要换成圆括号 ()：

```
first_word = ('direction', 'north')
second_word = ('verb', 'go')
sentence = [first_word, second_word]
```

这样我们就创建了一个 (TYPE, WORD) 组，让你识别出单词，并且对它执行指令。

这只是一个例子，不过最后做出来的样子也差不多。你接受用户输入，用 `split` 将其分隔成单词列表，然后分析这些单词，识别它们的类型，最后重新组成一个句子。

扫描输入

现在你要写的是词汇扫描器。这个扫描器会将用户的输入字符串当做参数，然后返回由多个 (TOKEN, WORD) 组成的一个列表，这个列表实现类似句子的功能。如果一个单词不在预定的词汇表中，那它返回时 WORD 应该还在，但 TOKEN 应该设置成一个专门的错误标记。这个错误标记将告诉用户哪里出错了。

有趣的地方来了。我不会告诉你这些该怎样做，但我会写一个“单元测试 (unit test)”，而你要把扫描器写出来，并保证单元测试能够正常通过。

“异常”和数字

有一件小事情我会先帮帮你，那就是数字转换。为了做到这一点，我们会作一点弊，使用“异常 (exceptions)”来做。“异常”指的是你运行某个函数时得到的错误。你的函数在碰到错误时，就会“提出 (raise)”一个“异常”，然后你就要去处理 (handle) 这个异常。假如你在 Python 里写了这些东西：

```
~/projects/simplegame $ python
Python 2.6.5 (r265:79063, Apr 16 2010, 13:57:41)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> int("hell")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'hell'
>>
```

这个 ValueError 就是 int() 函数抛出的一个异常。因为你给 int() 的参数不是一个数字。int() 函数其实也可以返回一个值来告诉你它碰到了错误，不过由于它只能返回整数值，所以很难做到这一点。它不能返回 -1，因为这也是一个数字。int() 没有纠结在它“究竟应该返回什么”上面，而是提出了一个叫做 ValueError 的异常，然后你只要处理这个异常就可以了。

处理异常的方法是使用 try 和 except 这两个关键字：

```
def convert_number(s):
    try:
        return int(s)
    except ValueError:
        return None
```

你把要试着运行的代码放到 try 的区段里，再将出错后要运行的代码放到 except 区段里。在这里，我们要试着调用 init() 去处理某个可能是数字的东西，如果中间出了错，我们就抓到这个错误，然后返回 None。

在你写的扫描器里面，你应该使用这个函数来测试某个东西是不是数字。做完这个检查，你就可以声明这个单词是一个错误单词了。

你应该测试的东西

这里是你应该使用的测试文件 tests/lexicon_tests.py：

```
from nose.tools import *
from ex48 import lexicon


def test_directions():
    assert_equal(lexicon.scan("north"), [('direction', 'north')])
    result = lexicon.scan("north south east")
    assert_equal(result, [('direction', 'north'),
                          ('direction', 'south'),
                          ('direction', 'east')])

```

```
def test_verbs():
    assert_equal(lexicon.scan("go"), [('verb', 'go')])
    result = lexicon.scan("go kill eat")
    assert_equal(result, [('verb', 'go'),
                          ('verb', 'kill'),
                          ('verb', 'eat')])


def test_stops():
    assert_equal(lexicon.scan("the"), [('stop', 'the')])
    result = lexicon.scan("the in of")
    assert_equal(result, [('stop', 'the'),
                          ('stop', 'in'),
                          ('stop', 'of')])


def test_nouns():
    assert_equal(lexicon.scan("bear"), [('noun', 'bear')])
    result = lexicon.scan("bear princess")
    assert_equal(result, [('noun', 'bear'),
                          ('noun', 'princess')])

def test_numbers():
    assert_equal(lexicon.scan("1234"), [('number', 1234)])
    result = lexicon.scan("3 91234")
    assert_equal(result, [('number', 3),
                          ('number', 91234)])


def test_errors():
    assert_equal(lexicon.scan("ASDFADFASDF"), [('error', 'ASDFADFASDF')])
    result = lexicon.scan("bear IAS princess")
    assert_equal(result, [('noun', 'bear'),
                          ('error', 'IAS'),
                          ('noun', 'princess')])
```

记住你要使用你的项目骨架来创建新项目，将这个测试用例写下来（不许复制粘贴！），然后编写你的扫描器，直至所有的测试都能通过。注意细节并确认结果一切工作良好。

设计的技巧

集中一次实现一个测试项目，尽量保持项目简单，只要把你的 `lexicon.py` 词汇表中所有的单词放那里就可以了。不要修改输入的单词表，不过你需要创建自己的新列表，里边包含你的语汇元组。另外，记得使用 `in` 关键字来检查这些语汇列表，以确认某个单词是否在你的语汇表中。

加分习题

1. 改进单元测试，让它覆盖到更多的语汇。
2. 向语汇列表添加更多的语汇，并且更新单元测试代码。
3. 让你的扫描器能够识别任意大小写的词汇。更新你的单元测试以确认其功能。
4. 找出另外一种转换为数字的方法。
5. 我的解决方案用了 37 行代码，你的是更长还是更短呢？

习题 49：创建句子

从我们这个小游戏的词汇扫描器中，我们应该可以得到类似下面的列表：

```
>>> from ex48 import lexicon
>>> print lexicon.scan("go north")
[('verb', 'go'), ('direction', 'north')]
>>> print lexicon.scan("kill the princess")
[('verb', 'kill'), ('stop', 'the'), ('noun', 'princess')]
>>> print lexicon.scan("eat the bear")
[('verb', 'eat'), ('stop', 'the'), ('noun', 'bear')]
>>> print lexicon.scan("open the door and smack the bear in the nose")
[('error', 'open'), ('stop', 'the'), ('noun', 'door'), ('error', 'and'),
('error', 'smack'), ('stop', 'the'), ('noun', 'bear'), ('stop', 'in'),
('stop', 'the'), ('error', 'nose')]
>>>
```

现在让我们把它转化成游戏可以使用的东西，也就是一个 Sentence 类。

如果你还记得学校学过的东西的话，一个句子是由这样的结构组成的：

主语 (Subject) + 谓语 (动词 Verb) + 宾语 (Object)

很显然实际的句子可能会比这复杂，而你可能已经在英语的语法课上面被折腾得够呛了。我们的目的，是将上面的元组列表转换为一个 Sentence 对象，而这个对象又包含主谓宾各个成员。

匹配 (Match) 和窥视 (Peek)

为了达到这个效果，你需要四样工具：

1. 循环访问元组列表的方法，这挺简单的。
2. 匹配我们的主谓宾设置中不同种类元组的方法。
3. 一个“窥视”潜在元组的方法，以便做决定时用到。
4. 跳过 (skip) 我们不在乎的内容的方法，例如形容词、冠词等没有用处的词汇。

我们使用 peek 函数来查看元组列表中的下一个成员，做匹配以后再对它做下一步动作。让我们先看看这个 peek 函数：

```
def peek(word_list):
    if word_list:
        word = word_list[0]
        return word[0]
    else:
        return None
```

很简单。再看看 match 函数：

```
def match(word_list, expecting):
    if word_list:
        word = word_list.pop(0)

        if word[0] == expecting:
            return word
        else:
            return None
    else:
        return None
```

还是很简单，最后我们看看 skip 函数:

```
def skip(word_list, word_type):
    while peek(word_list) == word_type:
        match(word_list, word_type)
```

以你现在的水平，你应该可以看出它们的功能来。确认自己真的弄懂了它们。

句子的语法

有了工具，我们现在可以从元组列表来构建句子 (Sentence) 对象了。我们的处理流程如下：

1. 使用 `peek` 识别下一个单词。
2. 如果这个单词和我们的语法匹配，我们就调用一个函数来处理这部分语法。假设函数的名字叫 `parse_subject` 好了。
3. 如果语法不匹配，我们就 `raise` 一个错误，接下来你会学到这方面的内容。
4. 全部分析完以后，我们应该能得到一个 Sentence 对象，然后可以将其应用在我们的游戏中。

演示这个过程最简单的方法是把代码展示给你让你阅读，不过这节习题有个不一样的要求，前面是我给你测试代码，你照着写出程序来，而这次是我给你的程序，而你要为它写出测试代码来。

以下就是我写的用来解析简单句子的代码，它使用了 `ex48.lexicon` 这个模组。

```
class ParserError(Exception):
    pass
```

```
class Sentence(object):

    def __init__(self, subject, verb, object):
        # remember we take ('noun','princess') tuples and convert them
        self.subject = subject[1]
        self.verb = verb[1]
        self.object = object[1]


def peek(word_list):
    if word_list:
        word = word_list[0]
        return word[0]
    else:
        return None


def match(word_list, expecting):
    if word_list:
        word = word_list.pop(0)

        if word[0] == expecting:
            return word
        else:
            return None
    else:
        return None


def skip(word_list, word_type):
    while peek(word_list) == word_type:
        match(word_list, word_type)


def parse_verb(word_list):
    skip(word_list, 'stop')

    if peek(word_list) == 'verb':
        return match(word_list, 'verb')
    else:
        raise ParserError("Expected a verb next.")
```

```
def parse_object(word_list):
    skip(word_list, 'stop')
    next = peek(word_list)

    if next == 'noun':
        return match(word_list, 'noun')
    if next == 'direction':
        return match(word_list, 'direction')
    else:
        raise ParserError("Expected a noun or direction next.")


def parse_subject(word_list, subj):
    verb = parse_verb(word_list)
    obj = parse_object(word_list)

    return Sentence(subj, verb, obj)


def parse_sentence(word_list):
    skip(word_list, 'stop')

    start = peek(word_list)

    if start == 'noun':
        subj = match(word_list, 'noun')
        return parse_subject(word_list, subj)
    elif start == 'verb':
        # assume the subject is the player then
        return parse_subject(word_list, ('noun', 'player'))
    else:
        raise ParserError("Must start with subject, object, or verb not: %s" % start)
```

关于异常 (Exception)

你已经简单学过关于异常的一些东西，但还没学过怎样抛出 (raise) 它们。这节的代码演示了如何 raise 前面定义的 `ParserError`。注意 `ParserError` 是一个定义为 `Exception` 类型的 class。另外要注意我们是怎样使用 `raise` 这个关键字来抛出异常的。

你的测试代码应该也要测试到这些异常，这个我也会演示给你如何实现。

你应该测试的东西

为《习题 49》写一个完整的测试方案，确认代码中所有的东西都能正常工作，其中异常的测试——输入一个错误的句子它会抛出一个异常来。

使用 `assert_raises` 这个函数来检查异常，在 nose 的文档里查看相关的内容，学着使用它写针对“执行失败”的测试，这也是测试很重要的一个方面。从 nose 文档中学会使用 `assert_raises`，以及一些别的函数。

写完测试以后，你应该就明白了这段程序的工作原理，而且也学会了如何为别人的程序写测试代码。相信我，这是一个非常有用的技能。

加分习题

1. 修改 `parse_` 函数（方法），将它们放到一个类里边，而不仅仅是独立的方法函数。这两种程序设计你喜欢哪一种呢？
2. 提高 parser 对于错误输入的抵御能力，这样即使用户输入了你预定义语汇之外的词语，你的程序也能正常运行下去。
3. 改进语法，让它可以处理更多的东西，例如数字。
4. 想想在游戏里你的 Sentence 类可以对用户输入做哪些有趣的事情。

习题 50：你的第一个网站

这节以及后面的习题中，你的任务是把前面创建的游戏做成网页版。这是本书的最后三个章节，这些内容对你来说难度会相当大，你要在上面花些时间才能做出来。在你开始这节练习以前，你必须已经成功地完成过了《习题 46》的内容，正确安装了 pip，而且学会了如何安装软件包以及如何创建项目骨架。如果你不记得这些内容，就回到《习题 46》重新复习一遍。

安装 lpthw.web

在创建你的第一个网页应用程序之前，你需要安装一个"Web 框架"，它的名字叫 lpthw.web。所谓的"框架"通常是指"让某件事情做起来更容易的软件包"。在网页应用的世界里，人们创建了各种各样的"网页框架"，用来解决他们在创建网站时碰到的问题，然后把这些解决方案用软件包的方式发布出来，这样你就可以利用它们引导创建你自己的项目了。

可选的框架类型有很多很多，不过在这里我们将使用 lpthw.web 框架。你可以先学会它，等到差不多的时候再去接触其它的框架，不过 lpthw.web 本身挺不错的，所以就算你一直使用也没关系。

使用 pip 安装 lpthw.web：

```
$ sudo pip install lpthw.web
[sudo] password for zedshaw:
Downloading/unpacking lpthw.web
  Running setup.py egg_info for package lpthw.web

Installing collected packages: lpthw.web
  Running setup.py install for lpthw.web

Successfully installed lpthw.web
Cleaning up...
```

以上是 Linux 和 Mac OSX 系统下的安装命令，如果你使用的是 Windows，那你只要把 sudo 去掉就可以了。如果你无法正常安装，请回到《习题 46》，确认自己学会了里边的内容。

Warning: 其他 Python 程序员会警告你说 `lpthw.web` 只是另外一个叫做 `web.py` 的 Web 框架的代码分支 (fork)，而 `web.py` 又包含了太多的“魔法 (magic)”在里边。如果他们这么说的话，你告诉他们 Google App Engine 最早用的就是 `web.py`，但没有一个 Python 程序员抱怨过它里边包含了太多的魔法，因为 Google 用它也没啥问题。如果 Google 觉得它可以，那它对你来说也不会差。所以还是回去继续学习吧，他们这些说法与其说是教导你，不如说是拿他们自己的教条束缚你，你还是忽略这些说法好了。

写一个简单的“Hello World”项目

现在你将做一个非常简单的“Hello World”项目出来，首先你要创建一个项目目录：

```
$ cd projects
$ mkdir gothonweb
$ cd gothonweb
$ mkdir bin gothonweb tests docs templates
$ touch gothonweb/__init__.py
$ touch tests/__init__.py
```

你最终的目的是把《习题 42》中的游戏做成一个 web 应用，所以你的项目名称叫做 `gothonweb`，不过在此之前，你需要创建一个最基本的 `lpthw.web` 应用，将下面的代码放到 `bin/app.py` 中：

```
import web

urls = (
  '/', 'index'
)

app = web.application(urls, globals())

class index:
    def GET(self):
        greeting = "Hello World"
        return greeting

if __name__ == "__main__":
    app.run()
```

然后使用下面的方法来运行这个 web 程序：

```
$ python bin/app.py
http://0.0.0.0:8080/
```

最后，使用你的网页浏览器，打开 URL `http://localhost:8080/`，你应该看到两样东西，首先是浏览器里显示了 `Hello, world!`，然后是你的命令行终端显示了如下的输出：

```
$ python bin/app.py
http://0.0.0.0:8080/
127.0.0.1:59542 - - [13/Jun/2011 11:44:43] "HTTP/1.1 GET /" - 200 OK
127.0.0.1:59542 - - [13/Jun/2011 11:44:43] "HTTP/1.1 GET /favicon.ico" - 404 Not Found
```

这些是 `lpthw.web` 打印出的 log 信息，从这些信息你可以看出服务器有在运行，而且能了解到程序在浏览器背后做了些什么事情。这些信息还有助于你发现程序的问题。例如在最后一行它告诉你浏览器试图获取 `/favicon.ico`，但是这个文件并不存在，因此它返回的状态码是 `404 Not Found`。

到这里，我还没有讲到任何 web 相关的工作原理，因为首先你需要完成准备工作，以便后面的学习能顺利进行，接下来的两节习题中会有详细的解释。我会要求你用各种方法把你的 lpthw.web 应用程序弄坏，然后再将其重新构建起来：这样做的目的是让你明白运行 `lpthw.web` 程序需要准备好哪些东西。

发生了什么事情？

在浏览器访问到你的网页应用程序时，发生了下面一些事情：

1. 浏览器通过网络连接到你自己的电脑，它的名字叫做 `localhost`，这是一个标准称谓，表示的谁就是网络中你自己的这台计算机，不管它实际名字是什么，你都可以使用 `localhost` 来访问。它使用到的网络端口是 `5000`。
2. 连接成功以后，浏览器对 `bin/app.py` 这个应用程序发出了 HTTP 请求 (request)，要求访问 URL `/`，这通常是一个网站的第一个 URL。
3. 在 `bin/app.py` 里，我们有一个列表，里边包含了 URL 和类的匹配关系。我们这里只定义了一组匹配，那就是 `'/'`, `'index'` 的匹配。它的含义是：如果有人使用浏览器访问 `/` 这一级目录，`lpthw.web` 将找到并加载 `class index`，从而用它处理这个浏览器请求。
4. 现在 `lpthw.web` 找到了 `class index`，然后针对这个类的一个实例调用了 `index.GET` 这个方法函数。该函数运行后返回了一个字符串，以供 `lpthw.web` 将其传递给浏览器。
5. 最后 `lpthw.web` 完成了对于浏览器请求的处理，将响应 (response) 回传给浏览器，于是你就看到了现在的页面。

确定你真的弄懂了这些，你需要画一个示意图，来理清信息是如何从浏览器传递到 `lpthw.web`，再到 `index.GET`，再回到你的浏览器的。

修正错误

第一步，把第 11 行的 `greeting` 变量赋值删掉，然后刷新浏览器。你应该会看到一个错误页面，你可以通过这一页丰富的错误信息看出你的程序崩溃的原因是什么。当然你已经知道出错的原因是 `greeting` 的赋值丢失了，不过 `lpthw.web` 还是会给你一个挺好的错误页面，让你能找到出错的具体位置。试试在这个错误页面上做以下操作：

1. 检查每一段 `Local vars` 输出（用鼠标点击它们），追踪里边提到的变量名称，以及它们是在哪些代码文件中用到的。
2. 阅读 `Request Information` 一节，看看里边哪些知识是你已经熟悉了的。Request 是浏览器发给你的 `gothonweb` 应用程序的信息。这些知识对于日常网页浏览没有什么用处，但现在你要学会这些东西，以便写出 web 应用程序来。
3. 试着把这个小程序的别的位置改错，探索一下会发生什么事情。‘‘lpthw.web‘‘ 的会把一些错误信息和堆栈跟踪 (stack trace) 信息显示在命令行终端，所以别忘了检查命令行终端的信息输出。

创建基本的模板文件

你已经试过用各种方法把这个 lpthw.web 程序改错，不过你有没有注意到“Hello World”不是一个好 HTML 网页呢？这是一个 web 应用，所以需要一个合适的 HTML 响应页面才对。为了达到这个目的，下一步你要做的是将“Hello World”以较大的绿色字体显示出来。

第一步是创建一个 `templates/index.html` 文件，内容如下：

```
$def with (greeting)

<html>
    <head>
        <title>Gothons Of Planet Percal #25</title>
    </head>
<body>

$if greeting:
    I just wanted to say <em style="color: green; font-size: 2em;">$greeting</em>.
$else:
    <em>Hello</em>, world!

</body>
</html>
```

如果你学过 HTML 的话，这些内容你看上去应该很熟悉。如果你没学过 HTML，那你应该去研究一下，试着用 HTML 写几个网页，从而知道它的工作原理。不过我们这里的 HTML 文件其实是一个“模板 (template)”，如果你向模板提供一些参数，`lpthw.web` 就会在模板中找到对应的位置，将参数的内容填充到模板中。例如每一个出现 `$greeting` 的位置，`$greeting` 的内容都会被替换成对应这个变量名的参数。

为了让你的 `bin/app.py` 处理模板，你需要写一写代码，告诉 `lpthw.web` 到哪里去找到模板进行加载，以及如何渲染 (render) 这个模板，按下面的方式修改你的 `app.py`：

```python
import web

urls = (
  '/', 'Index'
```

```
)

app = web.application(urls, globals())

render = web.template.render('templates/')

class Index(object):
    def GET(self):
        greeting = "Hello World"
        return render.index(greeting = greeting)

if __name__ == "__main__":
    app.run()
```

特别注意一下 render 这个新变量名，注意我修改了 index.GET 的最后一行，让它返回了 render.index() ，并且将 greeting 变量作为参数传递给了这个函数。

改好上面的代码后，刷新一下浏览器中的网页，你应该会看到一条和之前不同的绿色信息输出。你还可以在浏览器中通过“查看源文件 (View Source)”看到模板被渲染成了标准有效的 HTML 源代码。

这么讲也许有些太快了，我来详细解释一下模板的工作原理吧：

1. 在 bin/app.py 里面你添加了一个叫做 render 的新变量，它本身是一个 web.template.render 对象。
2. 你将 templates/ 作为参数传递给了这个对象，这样就让 render 知道了从哪里去加载模板文件。
3. 在你后面的代码中，当浏览器一如既往地触发了 index.GET 以后，它没有再返回简单的 greeting 字符串，取而代之的是你调用了 render.index，而且将问候语句作为一个变量传递给它。
4. 这个 render_template 函数可以说是一个“魔法函数”，它看到了你需要的是 index.html，于是就跑到 templates/ 目录下，找到名字为 index.html 的文件，然后就把它渲染 (render) 一遍（叫“转换一遍”也可以）。
5. 在 templates/index.html 文件中，你可以看到初始定义一行中说这个模板需要使用一个叫 greeting 的参数，这和函数定义中的格式差不多。另外和 Python 语法一样，模板文件是缩进敏感的，所以要确认自己弄对了缩进。
6. 最后，你让 templates/index.html 去检查 greeting 这个变量，如果这个变量存在的话，就打印出变量的内容，如果不存在的话，就会打印出一个默认的问候信息。

要深入理解这个过程，你可以修改 greeting 变量以及 HTML 模板的内容，看看会有什么效果。然后创建一个叫做 templates/foo.html 的模板，并且使用一个新的 render.foo 去渲染它。从这个过程你也可以看出，render 调用的函数名称只要跟 templates/ 下的 .html 文件名匹配到，这个 HTML 模板就可以被渲染到了。

加分习题

1. 到 http://webpy.org/ 阅读里边的文档，它其实和 `lpthw.web` 是同一个项目。
2. 实验一下你在上述网站看到的所有的东西，包括里边的代码示例。
3. 阅读以下 HTML5 和 CSS3 相关的东西，自己练习着写几个.html 和.css 文件。
4. 如果你有一个懂 Django 朋友可以帮你的话，你可以试着使用 Django 完成一下习题 50、51、52，看看结果会是什么样子的。

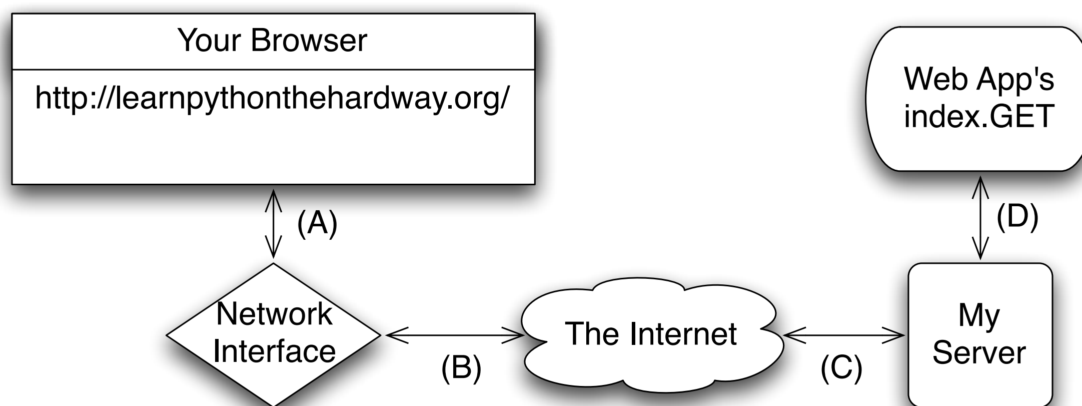
习题 51：从浏览器中获取输入

虽然能让浏览器显示“Hello World”是很有趣的一件事情，但是如果能让用户通过表单 (form) 向你的应用程序提交文本就更有趣了。这节习题中，我们将使用 form 改进你的 web 程序，并且将用户相关的信息保存到他们的“会话 (session)”中。

Web 的工作原理

该学点无趣的东西了。在创建 form 前你需要先多学一点关于 web 的工作原理。这里讲并不完整，但是相当准确，在你的程序出错是，它会帮你找到出错的原因。另外，如果你理解了 form 的应用，那么创建 form 对你来说就会更容易了。

我将以一个简单的图示讲起，它向你展示了 web 请求的各个不同的部分，以及信息传递的大致流程：



为了方便讲述 HTTP 请求 (request) 的流程，我在每条线上面加了字母标签以作区别。

1. 你在浏览器中输入网址 `http://learnpythonthehardway.org/`，然后浏览器会通过你的电脑的网络设备发出 request（线路 A）。
2. 你的 request 被传送到互联网（线路 B），然后再抵达远端服务器（线路 C），然后我的服务器将接受这个 request。

3. 我的服务器接受 request 后，我的 web 应用程序就去处理这个请求（线路 D），然后我的 Python 代码就会去运行 `index.GET` 这个“处理程序 (handler)”。
4. 在代码 `return` 的时候，我的 Python 服务器就会发出响应 (response)，这个响应会再通过线路 D 传递到你的浏览器。
5. 这个网站所在的服务器将响应由线路 D 获取，然后通过线路 C 传至互联网。
6. 响应通过互联网由线路 B 传至你的计算机，计算机的网卡再通过线路 A 将响应传给你的浏览器。
7. 最后，你的浏览器显示了这个响应的内容。

这段详解中用到了一些术语。你需要掌握这些术语，以便在谈论你的 web 应用时你能明白而且应用它们：

浏览器 (browser) 这是你几乎每天都会用到的软件。大部分人不知道它真正的原理，他们只会把它叫作“网”。它的作用其实是接收你输入到地址栏网址 (例如 http://learnpythonthehardway.org)，然后使用该信息向该网址对应的服务器提出请求 (request)。

地址 (address) 通常这是一个像 http://learnpythonthehardway.org/ 一样的 URL (Uniform Resource Locator，统一资源定位器)，它告诉浏览器该打开哪个网站。前面的 `http` 指出了你要使用的协议 (protocol)，这里我们用的是“超文本传输协议 (Hyper-Text Transport Protocol)”。你还可以试试 ftp://ibiblio.org/ ，这是一个“FTP 文件传输协议 (File Transport Protocol)”的例子。`learnpythonthehardway.org` 这部分是“主机名 (hostname)”，也就是一个便于人阅读和记忆的字串，主机名会被匹配到一串叫作“IP 地址”的数字上面，这个“IP 地址”就相当于网络中一台计算机的电话号码，通过这个号码可以访问到这台计算机。最后，URL 中还可以尾随一个“路径”，例如 http://learnpythonthehardway.org/book/ 中的 `/book/`，它对应的是服务器上的某个文件或者某些资源，通过访问这样的网址，你可以向服务器发出请求，然后获得这些资源。网站地址还有很多别的组成部分，不过这些是最主要的。

连接 (connection) 一旦浏览器知道了协议 (http)、服务器 (learnpythonthehardway.org)、以及要获得的资源，它就要去创建一个连接。这个过程中，浏览器让操作系统 (Operating System, OS) 打开计算机的一个“端口 (port)”（通常是 80 端口），端口准备好以后，操作系统会回传给你的程序一个类似文件的东西，它所做的事情就是通过网络传输和接收数据，让你的计算机和 learnpythonthehardway.org 这个网站所属的服务器之间实现数据交流。当你使用 http://localhost:8080/ 访问你自己的站点时，发生的事情其实是一样的，只不过这次你告诉了浏览器要访问的是你自己的计算机 (localhost)，要使用的端口不是默认的 80，而是 8080。你还可以直接访问 http://learnpythonthehardway.org:80/，这和不输入端口效果一样，因为 HTTP 的默认端口本来就是 80。

请求 (request) 你的浏览器通过你提供的地址建立了连接，现在它需要从远端服务器要到它（或你）想要的资源。如果你在 URL 的结尾加了 `/book/`，那你想要的就是/book/ 对应的文件或资源，大部分的服务器会直接为你调用/book/index.html 这个文件，不过我们就假装不存在好了。浏览器为了获得服务器上的资源，它需要向服务器发送一个“请求”。这里我就不讲细节了，为了得到服务器上的内容，你必须先向服务器发送一个请求才行。有意思的是，“资源”不一定非要是文件。例如当浏览器向你的应用程序提出请求的时候，服务器返回的其实是你的 Python 代码生成的一些东西。

服务器 (server) 服务器指的是浏览器另一端连接的计算机，它知道如何回应浏览器请求的文件和资源。大部分的 web 服务器只要发送文件就可以了，这也是服务器流量的主要部分。不过你学的是使用

Python 组建一个服务器，这个服务器知道如何接受请求，然后返回用 Python 处理过的字符串。当你使用这种处理方式时，你其实是假装把文件发给了浏览器，其实你用的都只是代码而已。就像你在《习题 50》中看到的，要构建一个“响应”其实也不需要多少代码。

响应 (response) 这就是你的服务器回复你的请求，发回至浏览器的 HTML，它里边可能有 css、javascript、或者图像等内容。以文件响应为例，服务器只要从磁盘读取文件，发送给浏览器就可以了，不过它还要将这些内容包在一个特别定义的“头部信息 (header)”中，这样浏览器就会知道它获取的是什么类型的内容。以你的 web 应用程序为例，你发送的其实还是一样的东西，包括 header 也一样，只不过这些数据是你用 Python 代码即时生成的。

这个可以算是你能在网上找到的关于浏览器如何访问网站的最快的快速课程了。这节课程应该可以帮你更容易地理解本节的习题，如果你还是不明白，就到处找资料多多了解这方面的信息，知道你明白为止。有一个很好的方法，就是你对照着上面的图示，将你在《习题 50》中创建的 web 程序中的内容分成几个部分，让其中的各部分对应到上面的图示。如果你可以正确地将程序的各部分对应到这个图示，你就大致开始明白它的工作原理了。

表单 (form) 的工作原理

熟悉“表单”最好的方法就是写一个可以接收表单数据的程序出来，然后看你可以对它做些什么。先将你的 `bin/app.py` 修改成下面的样子：

```python
import web

urls = (
  '/hello', 'Index'
)


app = web.application(urls, globals())

render = web.template.render('templates/')

class Index(object):
    def GET(self):
        form = web.input(name="Nobody")
        greeting = "Hello, %s" % form.name

        return render.index(greeting = greeting)

if __name__ == "__main__":
    app.run()
```

重启你的 web 程序（按 CTRL-C 后重新运行），确认它有运行起来，然后使用浏览器访问 `http://localhost:8080/hello`，这时浏览器应该会显示“I just wanted to say Hello, Nobody.”，接下来，

将浏览器的地址改成 http://localhost:8080/hello?name=Frank，然后你可以看到页面显示为“Hello, Frank.”，最后将 name=Frank 修改为你自己的名字，你就可以看到它对你说“Hello”了。

让我们研究一下你的程序里做过的修改。

1. 我们没有直接为 greeting 赋值，而是使用了 web.input 从浏览器获取数据。这个函数会将一组 key=value 的表述作为默认参数，解析你提供的 URL 中的 ?name=Frank 部分，然后返回一个对象，你可以通过这个对象方便地访问到表单的值。
2. 然后我通过 form 对象的 form.name 属性为 greeting 赋值，这句你应该已经熟悉了。
3. 其他的内容和以前是一样的，我们就不再分析了。

URL 中该还可以包含多个参数。将本例的 URL 改成这样子：http://localhost:8080/hello?name=Frank&greet=Hola。然后修改代码，让它去获取 form.name 和 form.greet，如下所示：

```
greeting = "%s, %s" % (form.greet, form.name)
```

修改完毕后，试着访问新的 URL。然后将 &greet=Hola 部分删除，看看你会得到什么样的错误信息。由于我们在 web.input(name="Nobody") 中没有为 greet 设定默认值，这样 greet 就变成了一个必须的参数，如果没有这个参数程序就会报错。现在修改一下你的程序，在 web.input 中为 greet 设一个默认值试试看。另外你还可以设 greet=None，这样你可以通过程序检查 greet 的值是否存在，然后提供一个比较好的错误信息出来，例如：

```
form = web.input(name="Nobody", greet=None)

if form.greet:
    greeting = "%s, %s" % (form.greet, form.name)
    return render.index(greeting = greeting)
else:
    return "ERROR: greet is required."
```

创建 HTML 表单

你可以通过 URL 参数实现表单提交，不过这样看上去有些丑陋，而且不方便一般人使用，你真正需要的是一个“POST 表单”，这是一种包含了 <form> 标签的特殊 HTML 文件。这种表单收集用户输入并将其传递给你的 web 程序，这和你上面实现的目的基本是一样的。

让我们来快速创建一个，从中你可以看出它的工作原理。你需要创建一个新的 HTML 文件，叫做 templates/hello_form.html：

```
<html>
    <head>
        <title>Sample Web Form</title>
    </head>
<body>
```

```
<h1>Fill Out This Form</h1>

<form action="/hello" method="POST">
    A Greeting: <input type="text" name="greet">
    <br/>
    Your Name: <input type="text" name="name">
    <br/>
    <input type="submit">
</form>

</body>
</html>
```

然后将 bin/app.py 改成这样：

```python
import web

urls = (
  '/hello', 'Index'
)

app = web.application(urls, globals())

render = web.template.render('templates/')

class Index(object):
    def GET(self):
        return render.hello_form()

    def POST(self):
        form = web.input(name="Nobody", greet="Hello")
        greeting = "%s, %s" % (form.greet, form.name)
        return render.index(greeting = greeting)

if __name__ == "__main__":
    app.run()
```

都写好以后，重启 web 程序，然后通过你的浏览器访问它。

这回你会看到一个表单，它要求你输入"一个问候语句 (A Greeting)"和"你的名字 (Your Name)"，等你输入完后点击"提交 (Submit)"按钮，它就会输出一个正常的问候页面，不过这一次你的 URL 还是 http://localhost:8080/hello，并没有添加参数进去。

在 hello_form.html 里面关键的一行是 <form action="/hello" method="POST">，它告诉你的浏览器以下内容：

1. 从表单中的各个栏位收集用户输入的数据。

2. 让浏览器使用一种 POST 类型的请求，将这些数据发送给服务器。这是另外一种浏览器请求，它会将表单栏位“隐藏”起来。

3. 将这个请求发送至 /hello URL，这是由 action="/hello" 告诉浏览器的。

你可以看到两段 <input> 标签的名字属性 (name) 和代码中的变量是对应的，另外我们在 class index 中使用的不再只是 GET 方法，而是另一个 POST 方法。

这个新程序的工作原理如下：

1. 浏览器访问到 web 程序的 /hello 目录，它发送了一个 GET 请求，于是我们的 index.GET 函数就运行并返回了 hello_form。

2. 你填好了浏览器的表单，然后浏览器依照 <form> 中的要求，将数据通过 POST 请求的方式发给 web 程序。

3. Web 程序运行了 index.POST 方法（不是 index.GET 方法）来处理这个请求。

4. 这个 index.POST 方法完成了它正常的功能，将 hello 页面返回，这里并没有新的东西，只是一个新函数名称而已。

作为练习，在 templates/index.html 中添加一个链接，让它指向 /hello，这样你可以反复填写并提交表单查看结果。确认你可以解释清楚这个链接的工作原理，以及它是如何让你实现在 templates/index.html 和 templates/hello_form.html 之间循环跳转的，还有就是要明白你新修改过的 Python 代码，你需要知道在什么情况下会运行到哪一部分代码。

创建布局模板 (layout template)

在你下一节练习创建游戏的过程中，你需要创建很多的小 HTML 页面。如果你每次都写一个完整的网页，你会很快感觉到厌烦的。幸运的是你可以创建一个“布局模板”，也就是一种提供了通用的头文件和脚注的外壳模板，你可以用它将你所有的其他网页包裹起来。好程序员会尽可能减少重复动作，所以要做一个好程序员，使用布局模板是很重要的。

将 templates/index.html 修改成这样：

```
$def with (greeting)

$if greeting:
    I just wanted to say <em style="color: green; font-size: 2em;">$greeting</em>.
$else:
    <em>Hello</em>, world!
```

然后把 templates/hello_form.html 修改成这样：

```
<h1>Fill Out This Form</h1>

<form action="/hello" method="POST">
    A Greeting: <input type="text" name="greet">
```

```
    <br/>
    Your Name: <input type="text" name="name">
    <br/>
    <input type="submit">
</form>
```

上面这些修改的目的，是将每一个页面顶部和底部的反复用到的“boilerplate”代码剥掉。这些被剥掉的代码会被放到一个单独的 `templates/layout.html` 文件中，从此以后，这些反复用到的代码就由 layout.html 来提供了。

上面的都改好以后，创建一个 `templates/layout.html` 文件，内容如下：

```
$def with (content)

<html>
<head>
    <title>Gothons From Planet Percal #25</title>
</head>
<body>

$:content

</body>
</html>
```

这个文件和普通的模板文件类似，不过其它的模板的内容将被传递给它，然后它会将其它模板的内容“包裹”起来。任何写在这里的内容多无需写在别的模板中了。你需要注意 `$:content` 的用法，这和其它的模板变量有些不同。

最后一步，就是将 `render` 对象改成这样：

```
render = web.template.render('templates/', base="layout")
```

这会告诉 `lpthw.web` 让它去使用 `templates/layout.html` 作为其它模板的基础模板。重启你的程序观察一下，然后试着用各种方法修改你的 layout 模板，不要修改你别的模板，看看输出会有什么样的变化。

为表单撰写自动测试代码

使用浏览器测试 web 程序是很容易的，只要点刷新按钮就可以了。不过毕竟我们是程序员嘛，如果我们可以写一些代码来测试我们的程序，为什么还要重复手动测试呢？接下来你要做的，就是为你的 web 程序写一个小测试。这会用到你在《习题 47》学过的一些东西，如果你不记得的话，可以回去复习一下。

为了让 Python 加载 `bin/app.py` 并进行测试，你需要先做一点准备工作。首先创建一个 `bin/__init__.py` 空文件，这样 Python 就会将 `bin/` 当作一个目录了。（在《习题 52》中你会去修改 ___init___.py，不过这是后话。）

我还为 lpthw.web 创建了一个简单的小函数，让你判断 (assert) web 程序的响应，这个函数有一个很合适的名字，就叫 assert_response。创建一个 tests/tools.py 文件，内容如下：

```python
from nose.tools import *
import re

def assert_response(resp, contains=None, matches=None, headers=None, status="200"):

    assert status in resp.status, "Expected response %r not in %r" % (status, resp.status)

    if status == "200":
        assert resp.data, "Response data is empty."

    if contains:
        assert contains in resp.data, "Response does not contain %r" % contains

    if matches:
        reg = re.compile(matches)
        assert reg.matches(resp.data), "Response does not match %r" % matches

    if headers:
        assert_equal(resp.headers, headers)
```

准备好这个文件以后，你就可以为你的 bin/app.py 写自动测试代码了。创建一个新文件，叫做 tests/app_tests.py，内容如下：

```python
from nose.tools import *
from bin.app import app
from tests.tools import assert_response

def test_index():
    # check that we get a 404 on the / URL
    resp = app.request("/")
    assert_response(resp, status="404")

    # test our first GET request to /hello
    resp = app.request("/hello")
    assert_response(resp)

    # make sure default values work for the form
    resp = app.request("/hello", method="POST")
    assert_response(resp, contains="Nobody")

    # test that we get expected values
    data = {'name': 'Zed', 'greet': 'Hola'}
    resp = app.request("/hello", method="POST", data=data)
```

```
    assert_response(resp, contains="Zed")
```

最后，使用 nosetests 运行测试脚本，然后测试你的 web 程序。

```
$ nosetests
.
----------------------------------------------------------------------
Ran 1 test in 0.059s

OK
```

这里我所做的，是将 `bin/app.py` 这个模块中的整个 web 程序都 import 进来，然后手动运行这个 web 程序。`lpthw.web` 有一个非常简单的 API 用来处理请求，看上去大致是这样子的：

```
app.request(localpart='/', method='GET', data=None, host='0.0.0.0:8080',
            headers=None, https=False)
```

你可以将 URL 作为第一个参数，然后你可以修改修改 request 的方法、form 的数据、以及 header 的内容，这样你无须启动 web 服务器，就可以使用自动测试来测试你的 web 程序了。

为了验证函数的响应，你需要使用 `tests.tools` 中定义的 `assert_response` 函数，用法属下：

```
assert_response(resp, contains=None, matches=None, headers=None, status="200")
```

把你调用 `app.request` 得到的响应传递给这个函数，然后将你要检查的内容作为参数传递给诶这个函数。你可以使用 `contains` 参数来检查响应中是否包含指定的值，使用 `status` 参数可以检查指定的响应状态。这个小函数其实包含了很多的信息，所以你还是自己研究一下的比较好。

在 `tests/app_tests.py` 自动测试脚本中，我首先确认 `/` 返回了一个“404 Not Found”响应，因为这个 URL 其实是不存在的。然后我检查了 `/hello` 在 `GET` 和 `POST` 两种请求的情况下都能正常工作。就算你没有弄明白测试的原理，这些测试代码应该是很好读懂的。

花一些时间研究一下这个最新版的 web 程序，重点研究一下自动测试的工作原理。确认你理解了将 `bin/app.py` 做为一个模块导入，然后进行自动化测试的流程。这是一个很重要的技巧，它会引导你学到更多东西。

加分习题

1. 阅读和 HTML 相关的更多资料，然后为你的表单设计一个更好的输出格式。你可以先在纸上设计出来，然后用 HTML 去实现它。
2. 这是一道难题，试着研究一下如何进行文件上传，通过网页上传一张图像，然后将其保存到磁盘中。
3. 更难的难题，找到 HTTP RFC 文件（讲述 HTTP 工作原理的技术文件），然后努力阅读一下。这是一篇很无趣的文档，不过偶尔你会用到里边的一些知识。
4. 又是一道难题，找人帮你设置一个 web 服务器，例如 Apache、Nginx、或者 thttpd。试着让服务器伺服一下你创建的.html 和.css 文件。如果失败了也没关系，web 服务器本来就都有点挫。

5. 完成上面的任务后休息一下，然后试着多创建一些 web 程序出来。你应该仔细阅读 `web.py` (它和 lpthw.web 是同一个程序) 中关于会话 (session) 的内容，这样你可以明白如何保持用户的状态信息。

习题 52: 创建你的 web 游戏

这本书马上就要结束了。本章的练习对你是一个真正的挑战。当你完成以后，你就可以算是一个能力不错的 Python 初学者了。为了进一步学习，你还需要多读一些书，多写一些程序，不过你已经具备进一步学习的技能了。接下来的学习就只是时间、动力、以及资源的问题了。

在本章习题中，我们不会去创建一个完整的游戏，取而代之的是我们会为《习题 42》中的游戏创建一个“引擎 (engine)”，让这个游戏能够在浏览器中运行起来。这会涉及到将《习题 42》中的游戏“重构 (refactor)”，将《习题 47》中的架构混合进来，添加自动测试代码，最后创建一个可以运行游戏的 web 引擎。

这是一节很庞大的习题。我预测你要花一周到一个月才能完成它。最好的方法是一点一点来，每晚上完成一点，在进行下一步之前确认上一步有正确完成。

重构《习题 42》的游戏

你已经在两个练习中修改了 `gothonweb` 项目，这节习题中你会再修改一次。这种修改的技术叫做“重构 (refactoring)”，或者用我喜欢的讲法来说，叫“修修补补 (fixing stuff)”。重构是一个编程术语，它指的是清理旧代码或者为旧代码添加新功能的过程。你其实已经做过这样的事情了，只不过不知道这个术语而已。这是写软件过程的第二个自然属性。

你在本节中要做的，是将《习题 47》中的可以测试的房间地图，以及《习题 42》中的游戏这两样东西归并到一起，创建一个新的游戏架构。游戏的内容不会发生变化，只不过我们会通过“重构”让它有一个更好的架构而已。

第一步是将 `ex47/game.py` 的内容复制到 `gothonweb/map.py`中，然后将 `tests/ex47_tests.py` 的内容复制到 `tests/map_tests.py` 中，然后再次运行 `nosetests`，确认他们还能正常工作。

Note: 从现在开始我不会再向你展示运行测试的输出了，我就假设你回去运行这些测试，而且知道怎样的输出是正确的。

将《习题 47》的代码拷贝完毕后，你就该开始重构它，让它包含《习题 42》中的地图。我一开始会把基本架构为你准备好，然后你需要去完成 `map.py` 和 `map_tests.py` 里边的内容。

首先要做的是使用 `Room` 类来构建基本的地图架构：

```python
class Room(object):

    def __init__(self, name, description):
        self.name = name
        self.description = description
        self.paths = {}

    def go(self, direction):
        return self.paths.get(direction, None)

    def add_paths(self, paths):
        self.paths.update(paths)


central_corridor = Room("Central Corridor",
"""
The Gothons of Planet Percal #25 have invaded your ship and destroyed
your entire crew.  You are the last surviving member and your last
mission is to get the neutron destruct bomb from the Weapons Armory,
put it in the bridge, and blow the ship up after getting into an
escape pod.

You're running down the central corridor to the Weapons Armory when
a Gothon jumps out, red scaly skin, dark grimy teeth, and evil clown costume
flowing around his hate filled body.  He's blocking the door to the
Armory and about to pull a weapon to blast you.
""")


laser_weapon_armory = Room("Laser Weapon Armory",
"""
Lucky for you they made you learn Gothon insults in the academy.
You tell the one Gothon joke you know:
Lbhe zbgure vf fb sng, jura fur fvgf nebhaq gur ubhfr, fur fvgf nebhaq gur ubhfr.
The Gothon stops, tries not to laugh, then busts out laughing and can't move.
While he's laughing you run up and shoot him square in the head
putting him down, then jump through the Weapon Armory door.

You do a dive roll into the Weapon Armory, crouch and scan the room
for more Gothons that might be hiding.  It's dead quiet, too quiet.
You stand up and run to the far side of the room and find the
neutron bomb in its container.  There's a keypad lock on the box
and you need the code to get the bomb out.  If you get the code
wrong 10 times then the lock closes forever and you can't
get the bomb.  The code is 3 digits.
```

```
""")


the_bridge = Room("The Bridge",
"""
The container clicks open and the seal breaks, letting gas out.
You grab the neutron bomb and run as fast as you can to the
bridge where you must place it in the right spot.

You burst onto the Bridge with the netron destruct bomb
under your arm and surprise 5 Gothons who are trying to
take control of the ship.  Each of them has an even uglier
clown costume than the last.  They haven't pulled their
weapons out yet, as they see the active bomb under your
arm and don't want to set it off.
""")


escape_pod = Room("Escape Pod",
"""
You point your blaster at the bomb under your arm
and the Gothons put their hands up and start to sweat.
You inch backward to the door, open it, and then carefully
place the bomb on the floor, pointing your blaster at it.
You then jump back through the door, punch the close button
and blast the lock so the Gothons can't get out.
Now that the bomb is placed you run to the escape pod to
get off this tin can.

You rush through the ship desperately trying to make it to
the escape pod before the whole ship explodes.  It seems like
hardly any Gothons are on the ship, so your run is clear of
interference.  You get to the chamber with the escape pods, and
now need to pick one to take.  Some of them could be damaged
but you don't have time to look.  There's 5 pods, which one
do you take?
""")


the_end_winner = Room("The End",
"""
You jump into pod 2 and hit the eject button.
The pod easily slides out into space heading to
the planet below.  As it flies to the planet, you look
back and see your ship implode then explode like a
```

```
bright star, taking out the Gothon ship at the same
time.  You won!
""")


the_end_loser = Room("The End",
"""
You jump into a random pod and hit the eject button.
The pod escapes out into the void of space, then
implodes as the hull ruptures, crushing your body
into jam jelly.
"""
)

escape_pod.add_paths({
    '2': the_end_winner,
    '*': the_end_loser
})

generic_death = Room("death", "You died.")

the_bridge.add_paths({
    'throw the bomb': generic_death,
    'slowly place the bomb': escape_pod
})

laser_weapon_armory.add_paths({
    '0132': the_bridge,
    '*': generic_death
})

central_corridor.add_paths({
    'shoot!': generic_death,
    'dodge!': generic_death,
    'tell a joke': laser_weapon_armory
})

START = central_corridor
```

你会发现我们的 Room 类和地图有一些问题：

1. 在进入一个房间以前会打印出一段文字作为房间的描述，我们需要将这些描述和每个房间关联起来，这样房间的次序就不会被打乱了，这对我们的游戏是一件好事。这些描述本来是在 if-else 结构中的，这是我们后面要修改的东西。

2. 原版游戏中我们使用了专门的代码来生成一些内容，例如炸弹的激活键码，舰舱的选择等，这次我们

做游戏时就先使用默认值好了，不过后面的加分习题里，我会要求你把这些功能再加到游戏中。

3. 我为所有的游戏中的失败结尾写了一个 generic_death，你需要去补全这个函数。你需要把原版游戏中所有的失败结尾都加进去，并确保代码能正确运行。
4. 我添加了一种新的转换模式，以 "*" 为标记，用来在游戏引擎中实现“catch-all”动作。

等你把上面的代码基本写好以后，接下来就是引导你继续写下去的自动测试的内容 tests/map_test.py 了：

```
from nose.tools import *
from gothonweb.map import *

def test_room():
    gold = Room("GoldRoom",
                """This room has gold in it you can grab. There's a
                door to the north.""")
    assert_equal(gold.name, "GoldRoom")
    assert_equal(gold.paths, {})

def test_room_paths():
    center = Room("Center", "Test room in the center.")
    north = Room("North", "Test room in the north.")
    south = Room("South", "Test room in the south.")

    center.add_paths({'north': north, 'south': south})
    assert_equal(center.go('north'), north)
    assert_equal(center.go('south'), south)

def test_map():
    start = Room("Start", "You can go west and down a hole.")
    west = Room("Trees", "There are trees here, you can go east.")
    down = Room("Dungeon", "It's dark down here, you can go up.")

    start.add_paths({'west': west, 'down': down})
    west.add_paths({'east': start})
    down.add_paths({'up': start})

    assert_equal(start.go('west'), west)
    assert_equal(start.go('west').go('east'), start)
    assert_equal(start.go('down').go('up'), start)

def test_gothon_game_map():
    assert_equal(START.go('shoot!'), generic_death)
    assert_equal(START.go('dodge!'), generic_death)

    room = START.go('tell a joke')
```

```
    assert_equal(room, laser_weapon_armory)
```

你在这部分练习中的任务是完成地图，并且让自动测试可以完整地检查过整个地图。这包括将所有的 `generic_death` 对象修正为游戏中实际的失败结尾。让你的代码成功运行起来，并让你的测试越全面越好。后面我们会对地图做一些修改，到时候这些测试将保证修改后的代码还可以正常工作。

会话 (session) 和用户跟踪

在你的 web 程序运行的某个位置，你需要追踪一些信息，并将这些信息和用户的浏览器关联起来。在 HTTP 协议的框架中，web 环境是“无状态 (stateless)”的，这意味着你的每一次请求和你其它的请求都是相互独立的。如果你请求了页面 A，输入了一些数据，然后点了一个页面 B 的链接，那你在页面 A 输入的数据就全部消失了。

解决这个问题的方法是为 web 程序建立一个很小的数据存储功能，给每个浏览器进程赋予一个独一无二的数字，用来跟踪浏览器所作的事情。这个存储通常用数据库或者存储在磁盘上的文件来实现。在 `lpthw.web` 这个小框架中实现这样的功能是很容易的，以下就是一个这样的例子：

```python
import web

web.config.debug = False

urls = (
    "/count", "count",
    "/reset", "reset"
)
app = web.application(urls, locals())
store = web.session.DiskStore('sessions')
session = web.session.Session(app, store, initializer={'count': 0})

class count:
    def GET(self):
        session.count += 1
        return str(session.count)

class reset:
    def GET(self):
        session.kill()
        return ""

if __name__ == "__main__":
    app.run()
```

为了实现这个功能，你需要创建一个 `sessions/` 文件夹作为程序的会话存储位置，创建好以后运行这个程序，然后检查 `/count` 页面，刷新一下这个页面，看计数会不会累加上去。关掉浏览器后，程序就会

“忘掉”之前的位置，这也是我们的游戏所需的功能。有一种方法可以让浏览器永远记住一些信息，不过这会让测试和开发变得更难。如果你回到 /reset/ 页面，然后再访问 /count 页面，你可以看到你的计数器被重置了，因为你已经把会话杀掉了。

你需要花点时间弄懂这段代码，注意会话开始时 count 的值是如何设为 0 的。另外再看看 sessions/ 下面的文件，看你能不能把它们打开。下面是我把一个 Python 会话打开并且解码的过程：

```
>>> import pickle
>>> import base64
>>> base64.b64decode(open("sessions/XXXXX").read())
"(dp1\nS'count'\np2\nI1\nsS'ip'\np3\nV127.0.0.1\np4\nsS'session_id'\np5\nS'XXXX'\np6\ns."
>>>
>>> x = base64.b64decode(open("sessions/XXXXX").read())
>>>
>>> pickle.loads(x)
{'count': 1, 'ip': u'127.0.0.1', 'session_id': 'XXXXX'}
```

所以会话其实就是使用 pickle 和 base64 这些库写到磁盘上的字典。存储和管理会话的方法很多，大概和 Python 的 web 框架那么多，所以了解它们的工作原理并不重要。当然如果你需要调试或者清空会话时，知道点原理还是有用的。

创建引擎

你应该已经写好了游戏地图和它的单元测试代码。现在我要求你制作一个简单的游戏引擎，用来让游戏中的各个房间运转起来，从玩家收集输入，并且记住玩家到了那一幕。我们将用到你刚学过的会话来制作一个简单的引擎，让它可以：

1. 为新用户启动新的游戏。
2. 将房间展示给用户。
3. 接受用户的输入。
4. 在游戏中处理用户的输入。
5. 显示游戏的结果，继续游戏的下一幕，知道玩家角色死亡为止。

为了创建这个引擎，你需要将我们久经考验的 bin/app.py 搬过来，创建一个功能完备的、基于会话的游戏引擎。这里的难点是我会先使用基本的 HTML 文件创建一个非常简单的版本，接下来将由你完成它，基本的引擎是这个样子的：

```
import web
from gothonweb import map

urls = (
  '/game', 'GameEngine',
  '/', 'Index',
)
```

```

app = web.application(urls, globals())

# little hack so that debug mode works with sessions
if web.config.get('_session') is None:
    store = web.session.DiskStore('sessions')
    session = web.session.Session(app, store,
                                  initializer={'room': None})
    web.config._session = session
else:
    session = web.config._session

render = web.template.render('templates/', base="layout")


class Index(object):
    def GET(self):
        # this is used to "setup" the session with starting values
        session.room = map.START
        web.seeother("/game")


class GameEngine(object):

    def GET(self):
        if session.room:
            return render.show_room(room=session.room)
        else:
            # why is there here? do you need it?
            return render.you_died()

    def POST(self):
        form = web.input(action=None)

        # there is a bug here, can you fix it?
        if session.room and form.action:
            session.room = session.room.go(form.action)

        web.seeother("/game")

if __name__ == "__main__":
    app.run()
```

这个脚本里你可以看到更多的新东西，不过了不起的事情是，整个基于网页的游戏引擎只要一个小文件就可以做到了。这段脚本里最有技术含量的事情就是将会话带回来的那几行，这对于调试模式下的代码重载

是必须的，否则每次你刷新网页，会话就会消失，游戏也不会再继续了。

在你运行 bin/app.py 之前，你需要修改 PYTHONPATH 环境变量。不知道什么是环境变量？为了运行一个最基本的 Python 程序，你就得学会环境变量，Python 的这一点确实有点挫。不过没办法，用 Python 的人就喜欢这样：

在你的命令行终端，输入下面的内容：

```
export PYTHONPATH=$PYTHONPATH:.
```

如果你用的是 Windows，那就输入以下内容:

```
set PYTHONPATH=%PYTHONPATH%;.
```

你只要针对每一个命令行会话界面输入一次就可以了，不过如果你运行 Python 代码时看到了 import 错误，那你就需要去执行一下上面的命令，或者也许是因为你上次执行的有错才导致 import 错误的。

接下来你需要删掉 `templates/hello_form.html` 和 `templates/index.html`，然后重新创建上面代码中提到的两个模板。这里是一个非常简单的 `templates/show_room.html` 供你参考：

```
$def with (room)

<h1> $room.name </h1>

<pre>
$room.description
</pre>

$if room.name == "death":
    <p><a href="/">Play Again?</a></p>
$else:
    <p>
    <form action="/game" method="POST">
        - <input type="text" name="action"> <input type="SUBMIT">
    </form>
    </p>
```

这就用来显示游戏中的房间的模板。接下来，你需要在用户跑到地图的边界时，用一个模板告诉用户他的角色的死亡信息，也就是 `templates/you_died.html` 这个模板：

```
<h1>You Died!</h1>

<p>Looks like you bit the dust.</p>
<p><a href="/">Play Again</a></p>
```

准备好了这些文件，你现在可以做下面的事情了：

1. 让测试代码 `tests/app_tests.py` 再次运行起来，这样你就可以去测试这个游戏。由于会话的存在，你可能顶多只能实现几次点击，不过你应该可以做出一些基本的测试来。

2. 删除 `sessions/*` 下的文件，再重新运行一遍游戏，确认游戏是从一开始运行起来的。
3. 执行 `python bin/app.py` 脚本，试玩一下你的游戏。

你需要和往常一样刷新和修正你的游戏，慢慢修改游戏的 HTML 文件和引擎，直到你实现游戏需要的所有功能为止。

你的期末考试

你有没有觉着我一下子给了你超多的信息呢？那就对了，我想要你在学习技能的同时可以有一些可以用来鼓捣的东西。为了完成这节习题，我将给你最后一套需要你自己完成的练习。你将注意到，到目前为止你写的游戏并不是很好，这只是你的第一版代码而已。你现在的任务是让游戏更加完善，实现下面的这些功能：

1. 修正代码中所有我提到和没提到的 bug，如果你发现了新的 bug，你可以告诉我。
2. 改进所有的自动测试，让你可以测试更多的内容，直到你可以不用浏览器就能测到所有的内容为止。
3. 让 HTML 页面看上去更美观一些。
4. 研究一下网页登录系统，为这个程序创建一个登录界面，这样人们就可以登录这个游戏，并且可以保存游戏高分。
5. 完成游戏地图，尽可能地把游戏做大，功能做全。
6. 给用户一个"帮助系统"，让他们可以查询每个房间里可以执行哪些命令。
7. 为你的游戏添加新功能，想到什么功能就添加什么功能。
8. 创建多个地图，让用户可以选择他们想要玩的一张来进行游戏。你的 `bin/app.py` 应该可以运行提供给它的任意的地图，这样你的引擎就可以支持多个不同的游戏。
9. 最后，使用你在习题 48 和 49 中学到的东西来创建一个更好的输入处理器。你手头已经有了大部分必要的代码，你只需要改进语法，让它和你的输入表单以及游戏引擎挂钩即可。

祝你好运！

下一步

现在还不能说你是一个程序员。这本书的目的相当于给你一个“编程棕带”。你已经了解了足够的编程基础，并且有能力阅读别的编程书籍了。读完这本书，你应该已经掌握了一些学习的方法，并且具备了该有的学习态度，这样你在阅读其他 Python 书籍时也许会更顺利，而且能学到更多东西。

在 http://learnpythonthehardway.org/ 网站列出了一些你可以进一步阅读的免费书籍，试着阅读它们，看看自己可以走多远。

或许，你现在已经可以开始鼓捣一些程序出来了。如果你手上有需要解决的问题，试着写个程序解决一下。你一开始写的东西可能很挫，不过这没有关系。以我为例，我在学每一种语言的初期都是很挫的。没有哪个初学者能写出完美的代码来，如果有人告诉你他有这本事，那他只是在厚着脸皮撒谎而已。

最后，记住学习编程是要投入时间的，你可能需要至少每天晚上练习几个小时。顺便告诉你，当你每晚学习 Python 的时候，我在努力学习弹吉他。我每天练习 2 到 4 小时，而且还在学习基本的音阶。

每个人都是某一方面的菜鸟。

老程序员的建议

你已经完成了这本书而且打算继续编程。也许这会成为你的一门职业，也许你只是作为业余爱好玩玩。无论如何，你都需要一些建议以保证你在正确的道路上继续前行，并且让这项新的爱好为你带来最大程度的享受。

我从事编程已经太长时间，长到对我来说编程已经是非常乏味的事情了。我写这本书的时候，已经懂得大约 20 种编程语言，而且可以在大约一天或者一个星期内学会一门编程语言 (取决于这门语言有多古怪)。现在对我来说编程这件事情已经很无聊，已经谈不上什么兴趣了。当然这不是说编程本身是一件无聊的事情，也不是说你以后也一定会这样觉得，这只是我个人在当前的感觉而已。

在这么久的旅程下来我的体会是：编程语言这东西并不重要，重要的是你用这些语言做的事情。事实上我一直知道这一点，不过以前我会周期性地被各种编程语言分神而忘记了这一点。现在我是永远不会忘记这一点了，你也不应该忘记这一点。

你学到和用到的编程语言并不重要。不要被围绕某一种语言的宗教把你扯进去，这只会让你忘掉了语言的真正目的，也就是作为你的工具来实现有趣的事情。

编程作为一项智力活动，是唯一一种能让你创建交互式艺术的艺术形式。你可以创建项目让别人使用，而且你可以间接地和使用者沟通。没有其他的艺术形式能做到如此程度的交互性。电影领着观众走向一个方向，绘画是不会动的。而代码却是双向互动的。

编程作为一项职业只是一般般有趣而已。编程可能是一份好工作，但如果你想赚更多的钱而且过得更快乐，你其实开一间快餐分店就可以了。你最好的选择是将你的编程技术作为你其他职业的秘密武器。

技术公司里边会编程的人多到一毛钱一打，根本得不到什么尊敬。而在生物学、医药学、政府部门、社会学、物理学、数学等行业领域从事编程的人就能得到足够的尊敬，而且你可以使用这项技能在这些领域做出令人惊异的成就。

当然，所有的这些建议都是没啥意义的。如果你跟着这本书学习写软件而且觉得很喜欢这件事情的话，那你完全可以将其当作一门职业去追求。你应该继续深入拓展这个近五十年来极少有人探索过的奇异而美妙的智力工作领域。若能从中得到乐趣当然就更好了。

最后我要说的是学习创造软件的过程会改变你而让你与众不同。不是说更好或更坏，只是不同了。你也许会发现因为你会写软件而人们对你的态度有些怪异，也许会用“怪人”这样的词来形容你。也许你会发现因为你会戳穿他们的逻辑漏洞而他们开始讨厌和你争辩。甚至你可能会发现有人因为你懂得计算机怎么工作而觉得你是个讨厌的怪人。

对于这些我只有一个建议: 让他们去死吧。这个世界需要更多的怪人，他们知道东西是怎么工作的而且

喜欢找到答案。当他们那样对你时，只要记住这是你的旅程，不是他们的。“与众不同”不是谁的错，告诉你“与众不同是一种错”的人只是嫉妒你掌握了他们做梦都不能想到的技能而已。

你会编程。他们不会。这真他妈的酷。

Indices and tables

- *search*